The embedonomicon

The embedonomicon walks you through the process of creating a `#![no_std]` application from scratch and through the iterative process of building architecture-specific functionality for Cortex-M microcontrollers.

Objectives

By reading this book you will learn

- How to build a `#![no_std]` application. This is much more complex than building a `#![no_std]` library because the target system may not be running an OS (or you could be aiming to build an OS!) and the program could be the only process running in the target (or the first one). In that case, the program may need to be customized for the target system.
- Tricks to finely control the memory layout of a Rust program. You'll learn about linkers, linker scripts and about the Rust features that let you control a bit of the ABI of Rust programs.
- A trick to implement default functionality that can be statically overridden (no runtime cost).

Target audience

This book mainly targets to two audiences:

- People that wish to bootstrap bare metal support for an architecture that the ecosystem doesn't yet support (e.g. Cortex-R as of Rust 1.28), or for an architecture that Rust just gained support for (e.g. maybe Xtensa some time in the future).
- People that are curious about the unusual implementation of *runtime* crates like `cortex-m-rt`, `msp430-rt` and `riscv-rt`.

Translations

This book has been translated by generous volunteers. If you would like your translation listed here, please open a PR to add it.

- Japanese (repository)
- Chinese (repository)

Requirements

This book is self contained. The reader doesn't need to be familiar with the Cortex-M architecture, nor is access to a Cortex-M microcontroller needed -- all the examples included in this book can be tested in QEMU. You will, however, need to install the following tools to run and inspect the examples in this book:

- All the code in this book uses the 2018 edition. If you are not familiar with the 2018 features and idioms check the `edition guide`.
- Rust 1.31 or a newer toolchain PLUS ARM Cortex-M compilation support.
- `cargo-binutils`. v0.1.4 or newer.
- `cargo-edit`.
- QEMU with support for ARM emulation. The `qemu-system-arm` program must be installed on your computer.
- GDB with ARM support.

Example setup

Instructions common to all OSes

```
$ # Rust toolchain
$ # If you start from scratch, get rustup from https://rustup.rs/
$ rustup default stable

$ # toolchain should be newer than this one
$ rustc -V
rustc 1.31.0 (abe02cefd 2018-12-04)

$ rustup target add thumbv7m-none-eabi

$ # cargo-binutils
$ cargo install cargo-binutils

$ rustup component add llvm-tools-preview
```

macOS

```
$ # arm-none-eabi-gdb
$ # you may need to run `brew tap Caskroom/tap` first
$ brew install --cask gcc-arm-embedded

$ # QEMU
$ brew install qemu
```

Ubuntu 16.04

```
$ # arm-none-eabi-gdb
$ sudo apt install gdb-arm-none-eabi

$ # QEMU
$ sudo apt install qemu-system-arm
```

Ubuntu 18.04 or Debian

```
$ # gdb-multiarch -- use `gdb-multiarch` when you wish to invoke gdb
$ sudo apt install gdb-multiarch

$ # QEMU
$ sudo apt install qemu-system-arm
```

Windows

- arm-none-eabi-gdb. The GNU Arm Embedded Toolchain includes GDB.
- QEMU

Installing a toolchain bundle from ARM (optional step) (tested on Ubuntu 18.04)

- With the late 2018 switch from GCC's linker to LLD for Cortex-M microcontrollers, gcc-arm-none-eabi is no longer required. But for those wishing to use the toolchain anyway, install from here and follow the steps outlined below:

```
$ tar xvjf gcc-arm-none-eabi-8-2018-q4-major-linux.tar.bz2
$ mv gcc-arm-none-eabi-<version_downloaded> <your_desired_path> # optional
$ export PATH=${PATH}:<path_to_arm_none_eabi_folder>/bin # add this line to
.bashrc to make persistent
```

The smallest `#![no_std]` program

In this section we'll write the smallest `#![no_std]` program that *compiles*.

What does `#![no_std]` mean?

`#![no_std]` is a crate level attribute that indicates that the crate will link to the `core` crate instead of the `std` crate, but what does this mean for applications?

The `std` crate is Rust's standard library. It contains functionality that assumes that the program will run on an operating system rather than *directly on the metal*. `std` also assumes that the operating system is a general purpose operating system, like the ones one would find in servers and desktops. For this reason, `std` provides a standard API over functionality one usually finds in such operating systems: Threads, files, sockets, a filesystem, processes, etc.

On the other hand, the `core` crate is a subset of the `std` crate that makes zero assumptions about the system the program will run on. As such, it provides APIs for language primitives like floats, strings and slices, as well as APIs that expose processor features like atomic operations and SIMD instructions. However it lacks APIs for anything that involves heap memory allocations and I/O.

For an application, `std` does more than just providing a way to access OS abstractions. `std` also takes care of, among other things, setting up stack overflow protection, processing command line arguments and spawning the main thread before a program's `main` function is invoked. A `#![no_std]` application lacks all that standard runtime, so it must initialize its own runtime, if any is required.

Because of these properties, a `#![no_std]` application can be the first and / or the only code that runs on a system. It can be many things that a standard Rust application can never be, for example:

- The kernel of an OS.
- Firmware.
- A bootloader.

The code

With that out of the way, we can move on to the smallest `#![no_std]` program that compiles:

```
$ cargo new --edition 2018 --bin app

$ cd app


$ # modify main.rs so it has these contents
$ cat src/main.rs


#![no_main]
#![no_std]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_panic: &PanicInfo<'_>) -> ! {
    loop {}
}
```

This program contains some things that you won't see in standard Rust programs:

The `#![no_std]` attribute which we have already extensively covered.

The `#![no_main]` attribute which means that the program won't use the standard `main` function as its entry point. At the time of writing, Rust's `main` interface makes some assumptions about the environment the program executes in: For example, it assumes the existence of command line arguments, so in general, it's not appropriate for `#![no_std]` programs.

The `#[panic_handler]` attribute. The function marked with this attribute defines the behavior of panics, both library level panics (`core::panic!`) and language level panics (out of bounds indexing).

This program doesn't produce anything useful. In fact, it will produce an empty binary.

```
$ # equivalent to `size target/thumbv7m-none-eabi/debug/app`
$ cargo size --target thumbv7m-none-eabi --bin app


   text    data     bss     dec     hex filename
      0       0       0       0       0 app
```

Before linking, the crate contains the panicking symbol.

```
$ cargo rustc --target thumbv7m-none-eabi -- --emit=obj

$ cargo nm -- target/thumbv7m-none-eabi/debug/deps/app-*.o | grep '[0-9]*
[^N] '
```

```
00000000 T rust_begin_unwind
```

However, it's our starting point. In the next section, we'll build something useful. But before continuing, let's set a default build target to avoid having to pass the `--target` flag to every Cargo invocation.

```
$ mkdir .cargo

$ # modify .cargo/config so it has these contents
$ cat .cargo/config
```

```
[build]
target = "thumbv7m-none-eabi"
```

eh_personality

If your configuration does not unconditionally abort on panic, which most targets for full operating systems don't (or if your custom target does not contain `"panic-strategy": "abort"`), then you must tell Cargo to do so or add an `eh_personality` function, which requires a nightly compiler. Here is Rust's documentation about it, and here is some discussion about it.

In your Cargo.toml, add:

```
[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"
```

Alternatively, declare the `eh_personality` function. A simple implementation that does not do anything special when unwinding is as follows:

```
#![feature(lang_items)]

#[lang = "eh_personality"]
extern "C" fn eh_personality() {}
```

You will receive the error `language item required, but not found: 'eh_personality'` if not included.

Memory layout

The next step is to ensure the program has the right memory layout so that the target system will be able to execute it. In our example, we'll be working with a virtual Cortex-M3 microcontroller: the LM3S6965. Our program will be the only process running on the device so it must also take care of initializing the device.

Background information

Cortex-M devices require a vector table to be present at the start of their code memory region. The vector table is an array of pointers; the first two pointers are required to boot the device, the rest of the pointers are related to exceptions. We'll ignore them for now.

Linkers decide the final memory layout of programs, but we can use linker scripts to have some control over it. The control granularity that linker scripts give us over the layout is at the level of *sections*. A section is a collection of *symbols* laid out in contiguous memory. Symbols, in turn, can be data (a static variable), or instructions (a Rust function).

Every symbol has a name assigned by the compiler. As of Rust 1.28 , the names that the Rust compiler assigns to symbols are of the form: `_ZN5krate6module8function17he1dfc17c86fe16daE` , which demangles to `krate::module::function::he1dfc17c86fe16da` where `krate::module::function` is the path of the function or variable and `he1dfc17c86fe16da` is some sort of hash. The Rust compiler will place each symbol into its own unique section; for example the symbol mentioned before will be placed in a section named `.text._ZN5krate6module8function17he1dfc17c86fe16daE` .

These compiler generated symbol and section names are not guaranteed to remain constant across different releases of the Rust compiler. However, the language lets us control symbol names and section placement via these attributes:

- `#[export_name = "foo"]` sets the symbol name to `foo` .
- `#[no_mangle]` means: use the function or variable name (not its full path) as its symbol name. `#[no_mangle] fn bar()` will produce a symbol named `bar` .
- `#[link_section = ".bar"]` places the symbol in a section named `.bar` .

With these attributes, we can expose a stable ABI of the program and use it in the linker script.

The Rust side

As mentioned above, for Cortex-M devices, we need to populate the first two entries of the vector table. The first one, the initial value for the stack pointer, can be populated using only the linker script. The second one, the reset vector, needs to be created in Rust code and placed correctly using the linker script.

The reset vector is a pointer into the reset handler. The reset handler is the function that the device will execute after a system reset, or after it powers up for the first time. The reset handler is always the first stack frame in the hardware call stack; returning from it is undefined behavior as there's no other stack frame to return to. We can enforce that the reset handler never returns by making it a divergent function, which is a function with signature `fn(/* .. */) -> !`.

```rust
#[no_mangle]
pub unsafe extern "C" fn Reset() -> ! {
    let _x = 42;

    // can't return so we go into an infinite loop here
    loop {}
}

// The reset vector, a pointer into the reset handler
#[link_section = ".vector_table.reset_vector"]
#[no_mangle]
pub static RESET_VECTOR: unsafe extern "C" fn() -> ! = Reset;
```

The hardware expects a certain format here, to which we adhere by using `extern "C"` to tell the compiler to lower the function using the C ABI, instead of the Rust ABI, which is unstable.

To refer to the reset handler and reset vector from the linker script, we need them to have a stable symbol name so we use `#[no_mangle]`. We need fine control over the location of `RESET_VECTOR`, so we place it in a known section, `.vector_table.reset_vector`. The exact location of the reset handler itself, `Reset`, is not important. We just stick to the default compiler generated section.

The linker will ignore symbols with internal linkage (also known as internal symbols) while traversing the list of input object files, so we need our two symbols to have external linkage. The only way to make a symbol external in Rust is to make its corresponding item public (`pub`) and *reachable* (no private module between the item and the root of the crate).

The linker script side

A minimal linker script that places the vector table in the correct location is shown below.

Let's walk through it.

```
$ cat link.x
```

```
/* Memory layout of the LM3S6965 microcontroller */
/* 1K = 1 KiBi = 1024 bytes */
MEMORY
{
  FLASH : ORIGIN = 0x00000000, LENGTH = 256K
  RAM : ORIGIN = 0x20000000, LENGTH = 64K
}

/* The entry point is the reset handler */
ENTRY(Reset);

EXTERN(RESET_VECTOR);

SECTIONS
{
  .vector_table ORIGIN(FLASH) :
  {
    /* First entry: initial Stack Pointer value */
    LONG(ORIGIN(RAM) + LENGTH(RAM));

    /* Second entry: reset vector */
    KEEP(*(.vector_table.reset_vector));
  } > FLASH

  .text :
  {
    *(.text .text.*);
  } > FLASH

  /DISCARD/ :
  {
    *(.ARM.exidx .ARM.exidx.*);
  }
}
```

MEMORY

This section of the linker script describes the location and size of blocks of memory in the target. Two memory blocks are defined: `FLASH` and `RAM`; they correspond to the physical memory available in the target. The values used here correspond to the LM3S6965 microcontroller.

ENTRY

Here we indicate to the linker that the reset handler, whose symbol name is `Reset`, is the *entry point* of the program. Linkers aggressively discard unused sections. Linkers consider the entry point and functions called from it as *used* so they won't discard them. Without this line, the linker would discard the `Reset` function and all subsequent functions called from it.

`EXTERN`

Linkers are lazy; they will stop looking into the input object files once they have found all the symbols that are recursively referenced from the entry point. `EXTERN` forces the linker to look for `EXTERN`'s argument even after all other referenced symbols have been found. As a rule of thumb, if you need a symbol that's not called from the entry point to always be present in the output binary, you should use `EXTERN` in conjunction with `KEEP`.

`SECTIONS`

This part describes how sections in the input object files (also known as *input sections*) are to be arranged in the sections of the output object file (also known as output sections) or if they should be discarded. Here we define two output sections:

```
  .vector_table ORIGIN(FLASH) : { /* .. */ } > FLASH
```

`.vector_table` contains the vector table and is located at the start of `FLASH` memory.

```
  .text : { /* .. */ } > FLASH
```

`.text` contains the program subroutines and is located somewhere in `FLASH`. Its start address is not specified, but the linker will place it after the previous output section, `.vector_table`.

The output `.vector_table` section contains:

```
    /* First entry: initial Stack Pointer value */
    LONG(ORIGIN(RAM) + LENGTH(RAM));
```

We'll place the (call) stack at the end of RAM (the stack is *full descending*; it grows towards smaller addresses) so the end address of RAM will be used as the initial Stack Pointer (SP) value. That address is computed in the linker script itself using the information we entered for the `RAM` memory block.

```
    /* Second entry: reset vector */
    KEEP(*(.vector_table.reset_vector));
```

Next, we use `KEEP` to force the linker to insert all input sections named `.vector_table.reset_vector` right after the initial SP value. The only symbol located in that section is `RESET_VECTOR`, so this will effectively place `RESET_VECTOR` second in the vector table.

The output `.text` section contains:

```
    *(.text .text.*);
```

This includes all the input sections named `.text` and `.text.*`. Note that we don't use `KEEP` here to let the linker discard unused sections.

Finally, we use the special `/DISCARD/` section to discard

```
    *(.ARM.exidx .ARM.exidx.*);
```

input sections named `.ARM.exidx.*`. These sections are related to exception handling but we are not doing stack unwinding on panics and they take up space in Flash memory, so we just discard them.

Putting it all together

Now we can link the application. For reference, here's the complete Rust program:

```
#![no_main]
#![no_std]

use core::panic::PanicInfo;

// The reset handler
#[no_mangle]
pub unsafe extern "C" fn Reset() -> ! {
    let _x = 42;

    // can't return so we go into an infinite loop here
    loop {}
}

// The reset vector, a pointer into the reset handler
#[link_section = ".vector_table.reset_vector"]
#[no_mangle]
pub static RESET_VECTOR: unsafe extern "C" fn() -> ! = Reset;

#[panic_handler]
fn panic(_panic: &PanicInfo<'_>) -> ! {
    loop {}
}
```

We have to tweak the linker process to make it use our linker script. This is done passing the `-C link-arg` flag to `rustc`. This can be done with `cargo-rustc` or `cargo-build`.

IMPORTANT: Make sure you have the `.cargo/config` file that was added at the end of the last section before running this command.

Using the `cargo-rustc` subcommand:

```
$ cargo rustc -- -C link-arg=-Tlink.x
```

Or you can set the rustflags in `.cargo/config` and continue using the `cargo-build` subcommand. We'll do the latter because it better integrates with `cargo-binutils`.

```
# modify .cargo/config so it has these contents
$ cat .cargo/config
```

```
[target.thumbv7m-none-eabi]
rustflags = ["-C", "link-arg=-Tlink.x"]

[build]
target = "thumbv7m-none-eabi"
```

The `[target.thumbv7m-none-eabi]` part says that these flags will only be used when cross compiling to that target.

Inspecting it

Now let's inspect the output binary to confirm the memory layout looks the way we want (this requires `cargo-binutils`):

```
$ cargo objdump --bin app -- -d --no-show-raw-insn
```

```


app:    file format elf32-littlearm

Disassembly of section .text:

<Reset>:
                sub     sp, #4
                movs    r0, #42
                str     r0, [sp]
                b       0x10 <Reset+0x8>        @ imm = #-2
                b       0x10 <Reset+0x8>        @ imm = #-4
```

This is the disassembly of the `.text` section. We see that the reset handler, named

`Reset`, is located at address `0x8`.

```
$ cargo objdump --bin app -- -s --section .vector_table
```

```
app:    file format elf32-littlearm
Contents of section .vector_table:
 0000 00000120 09000000                    ... ....
```

This shows the contents of the `.vector_table` section. We can see that the section starts at address `0x0` and that the first word of the section is `0x2001_0000` (the `objdump` output is in little endian format). This is the initial SP value and matches the end address of RAM. The second word is `0x9`; this is the *thumb mode* address of the reset handler. When a function is to be executed in thumb mode the first bit of its address is set to 1.

Testing it

This program is a valid LM3S6965 program; we can execute it in a virtual microcontroller (QEMU) to test it out.

```
$ # this program will block
$ qemu-system-arm \
      -cpu cortex-m3 \
      -machine lm3s6965evb \
      -gdb tcp::3333 \
      -S \
      -nographic \
      -kernel target/thumbv7m-none-eabi/debug/app
```

```
$ # on a different terminal
$ arm-none-eabi-gdb -q target/thumbv7m-none-eabi/debug/app
Reading symbols from target/thumbv7m-none-eabi/debug/app...done.

(gdb) target remote :3333
Remote debugging using :3333
Reset () at src/main.rs:8
8       pub unsafe extern "C" fn Reset() -> ! {

(gdb) # the SP has the initial value we programmed in the vector table
(gdb) print/x $sp
$1 = 0x20010000

(gdb) step
9           let _x = 42;

(gdb) step
12          loop {}

(gdb) # next we inspect the stack variable `_x`
(gdb) print _x
$2 = 42

(gdb) print &_x
$3 = (i32 *) 0x2000fffc

(gdb) quit
```

A `main` interface

We have a minimal working program now, but we need to package it in a way that the end user can build safe programs on top of it. In this section, we'll implement a `main` interface like the one standard Rust programs use.

First, we'll convert our binary crate into a library crate:

```
$ mv src/main.rs src/lib.rs
```

And then rename it to `rt` which stands for "runtime".

```
$ sed -i s/app/rt/ Cargo.toml

$ head -n4 Cargo.toml
```

```
[package]
edition = "2018"
name = "rt" # <-
version = "0.1.0"
```

The first change is to have the reset handler call an external `main` function:

```
$ head -n13 src/lib.rs
```

```
#![no_std]

use core::panic::PanicInfo;

// CHANGED!
#[no_mangle]
pub unsafe extern "C" fn Reset() -> ! {
    extern "Rust" {
        fn main() -> !;
    }

    main()
}
```

We also drop the `#![no_main]` attribute as it has no effect on library crates.

> There's an orthogonal question that arises at this stage: Should the `rt` library provide a standard panicking behavior, or should it *not* provide a `#[panic_handler]` function and leave the end user to choose the panicking behavior? This document won't delve into that question and for simplicity will leave the dummy

`#[panic_handler]` function in the `rt` crate. However, we wanted to inform the reader that there are other options.

The second change involves providing the linker script we wrote before to the application crate. The linker will search for linker scripts in the library search path (`-L`) and in the directory from which it's invoked. The application crate shouldn't need to carry around a copy of `link.x` so we'll have the `rt` crate put the linker script in the library search path using a build script.

```
$ # create a build.rs file in the root of `rt` with these contents
$ cat build.rs
```

```
use std::{env, error::Error, fs::File, io::Write, path::PathBuf};

fn main() -> Result<(), Box<dyn Error>> {
    // build directory for this crate
    let out_dir = PathBuf::from(env::var_os("OUT_DIR").unwrap());

    // extend the library search path
    println!("cargo:rustc-link-search={}", out_dir.display());

    // put `link.x` in the build directory
    File::create(out_dir.join("link.x"))?.write_all(include_bytes!
("link.x"))?;

    Ok(())
}
```

Now the user can write an application that exposes the `main` symbol and link it to the `rt` crate. The `rt` will take care of giving the program the right memory layout.

```
$ cd ..

$ cargo new --edition 2018 --bin app

$ cd app

$ # modify Cargo.toml to include the `rt` crate as a dependency
$ tail -n2 Cargo.toml
```

```
[dependencies]
rt = { path = "../rt" }
```

```
$ # copy over the config file that sets a default target and tweaks the
linker invocation
$ cp -r ../rt/.cargo .

$ # change the contents of `main.rs` to
$ cat src/main.rs
```

```
#![no_std]
#![no_main]

extern crate rt;

#[no_mangle]
pub fn main() -> ! {
    let _x = 42;

    loop {}
}
```

The disassembly will be similar but will now include the user `main` function.

```
$ cargo objdump --bin app -- -d --no-show-raw-insn
```

```


app:    file format elf32-littlearm

Disassembly of section .text:

<main>:
                sub     sp, #4
                movs    r0, #42
                str     r0, [sp]
                b       0x10 <main+0x8>         @ imm = #-2
                b       0x10 <main+0x8>         @ imm = #-4

<Reset>:
                push    {r7, lr}
                mov     r7, sp
                bl      0x8 <main>              @ imm = #-18
                trap
```

Making it type safe

The `main` interface works, but it's easy to get it wrong. For example, the user could write `main` as a non-divergent function, and they would get no compile time error and undefined behavior (the compiler will misoptimize the program).

We can add type safety by exposing a macro to the user instead of the symbol interface.

In the `rt` crate, we can write this macro:

```
$ tail -n12 ../rt/src/lib.rs
```

```rust
#[macro_export]
macro_rules! entry {
    ($path:path) => {
        #[export_name = "main"]
        pub unsafe fn __main() -> ! {
            // type check the given path
            let f: fn() -> ! = $path;

            f()
        }
    }
}
```

Then the application writers can invoke it like this:

```
$ cat src/main.rs
```

```rust
#![no_std]
#![no_main]

use rt::entry;

entry!(main);

fn main() -> ! {
    let _x = 42;

    loop {}
}
```

Now the author will get an error if they change the signature of `main` to be non divergent function, e.g. `fn()`.

Life before main

`rt` is looking good but it's not feature complete! Applications written against it can't use `static` variables or string literals because `rt`'s linker script doesn't define the standard `.bss`, `.data` and `.rodata` sections. Let's fix that!

The first step is to define these sections in the linker script:

```
$ # showing just a fragment of the file
$ sed -n 25,46p ../rt/link.x


  .text :
  {
    *(.text .text.*);
  } > FLASH

  /* NEW! */
  .rodata :
  {
    *(.rodata .rodata.*);
  } > FLASH

  .bss :
  {
    *(.bss .bss.*);
  } > RAM

  .data :
  {
    *(.data .data.*);
  } > RAM

  /DISCARD/ :
```

They just re-export the input sections and specify in which memory region each output section will go.

With these changes, the following program will compile:

```
#![no_std]
#![no_main]

use rt::entry;

entry!(main);

static RODATA: &[u8] = b"Hello, world!";
static mut BSS: u8 = 0;
static mut DATA: u16 = 1;

fn main() -> ! {
    let _x = RODATA;
    let _y = unsafe { &BSS };
    let _z = unsafe { &DATA };

    loop {}
}
```

However if you run this program on real hardware and debug it, you'll observe that the `static` variables `BSS` and `DATA` don't have the values `0` and `1` by the time `main` has been reached. Instead, these variables will have junk values. The problem is that the contents of RAM are random after powering up the device. You won't be able to observe this effect if you run the program in QEMU.

As things stand if your program reads any `static` variable before performing a write to it then your program has undefined behavior. Let's fix that by initializing all `static` variables before calling `main`.

We'll need to tweak the linker script a bit more to do the RAM initialization:

```
$ # showing just a fragment of the file
$ sed -n 25,52p ../rt/link.x
```

```

  .text :
  {
    *(.text .text.*);
  } > FLASH

  /* CHANGED! */
  .rodata :
  {
    *(.rodata .rodata.*);
  } > FLASH

  .bss :
  {
    _sbss = .;
    *(.bss .bss.*);
    _ebss = .;
  } > RAM

  .data : AT(ADDR(.rodata) + SIZEOF(.rodata))
  {
    _sdata = .;
    *(.data .data.*);
    _edata = .;
  } > RAM

  _sidata = LOADADDR(.data);

  /DISCARD/ :
```

Let's go into the details of these changes:

```
    _sbss = .;
```

```
    _ebss = .;
```

```
    _sdata = .;
    _edata = .;
```

We associate symbols to the start and end addresses of the `.bss` and `.data` sections, which we'll later use from Rust code.

```
  .data : AT(ADDR(.rodata) + SIZEOF(.rodata))
```

We set the Load Memory Address (LMA) of the `.data` section to the end of the `.rodata` section. The `.data` contains `static` variables with a non-zero initial value; the Virtual Memory Address (VMA) of the `.data` section is somewhere in RAM -- this is where the `static` variables are located. The initial values of those `static` variables, however, must be allocated in non volatile memory (Flash); the LMA is where in Flash those initial values are stored.

```
  _sidata = LOADADDR(.data);
```

Finally, we associate a symbol to the LMA of `.data`.

On the Rust side, we zero the `.bss` section and initialize the `.data` section. We can reference the symbols we created in the linker script from the Rust code. The *addresses*[1] of these symbols are the boundaries of the `.bss` and `.data` sections.

The updated reset handler is shown below:

```
$ head -n32 ../rt/src/lib.rs
```

```
#![no_std]

use core::panic::PanicInfo;
use core::ptr;

#[no_mangle]
pub unsafe extern "C" fn Reset() -> ! {
    // NEW!
    // Initialize RAM
    extern "C" {
        static mut _sbss: u8;
        static mut _ebss: u8;

        static mut _sdata: u8;
        static mut _edata: u8;
        static _sidata: u8;
    }

    let count = &_ebss as *const u8 as usize - &_sbss as *const u8 as usize;
    ptr::write_bytes(&mut _sbss as *mut u8, 0, count);

    let count = &_edata as *const u8 as usize - &_sdata as *const u8 as
usize;
    ptr::copy_nonoverlapping(&_sidata as *const u8, &mut _sdata as *mut u8,
count);

    // Call user entry point
    extern "Rust" {
        fn main() -> !;
    }

    main()
}
```

Now end users can directly and indirectly make use of `static` variables without running into undefined behavior!

> In the code above we performed the memory initialization in a bytewise fashion. It's possible to force the `.bss` and `.data` sections to be aligned to, say, 4 bytes. This fact can then be used in the Rust code to perform the initialization wordwise while omitting alignment checks. If you are interested in learning how this can be achieved check the `cortex-m-rt` crate.

1 The fact that the addresses of the linker script symbols must be used here can be confusing and unintuitive. An elaborate explanation for this oddity can be found here.

Exception handling

During the "Memory layout" section, we decided to start out simple and leave out handling of exceptions. In this section, we'll add support for handling them; this serves as an example of how to achieve compile time overridable behavior in stable Rust (i.e. without relying on the unstable `#[linkage = "weak"]` attribute, which makes a symbol weak).

Background information

In a nutshell, *exceptions* are a mechanism the Cortex-M and other architectures provide to let applications respond to asynchronous, usually external, events. The most prominent type of exception, that most people will know, is the classical (hardware) interrupt.

The Cortex-M exception mechanism works like this: When the processor receives a signal or event associated to a type of exception, it suspends the execution of the current subroutine (by stashing the state in the call stack) and then proceeds to execute the corresponding exception handler, another subroutine, in a new stack frame. After finishing the execution of the exception handler (i.e. returning from it), the processor resumes the execution of the suspended subroutine.

The processor uses the vector table to decide what handler to execute. Each entry in the table contains a pointer to a handler, and each entry corresponds to a different exception type. For example, the second entry is the reset handler, the third entry is the NMI (Non Maskable Interrupt) handler, and so on.

As mentioned before, the processor expects the vector table to be at some specific location in memory, and each entry in it can potentially be used by the processor at runtime. Hence, the entries must always contain valid values. Furthermore, we want the `rt` crate to be flexible so the end user can customize the behavior of each exception handler. Finally, the vector table resides in read only memory, or rather in not easily modified memory, so the user has to register the handler statically, rather than at runtime.

To satisfy all these constraints, we'll assign a *default* value to all the entries of the vector table in the `rt` crate, but make these values kind of *weak* to let the end user override them at compile time.

Rust side

Let's see how all this can be implemented. For simplicity, we'll only work with the first 16 entries of the vector table; these entries are not device specific so they have the same function on any kind of Cortex-M microcontroller.

The first thing we'll do is create an array of vectors (pointers to exception handlers) in the `rt` crate's code:

```
$ sed -n 56,91p ../rt/src/lib.rs
```

```rust
pub union Vector {
    reserved: u32,
    handler: unsafe extern "C" fn(),
}

extern "C" {
    fn NMI();
    fn HardFault();
    fn MemManage();
    fn BusFault();
    fn UsageFault();
    fn SVCall();
    fn PendSV();
    fn SysTick();
}

#[link_section = ".vector_table.exceptions"]
#[no_mangle]
pub static EXCEPTIONS: [Vector; 14] = [
    Vector { handler: NMI },
    Vector { handler: HardFault },
    Vector { handler: MemManage },
    Vector { handler: BusFault },
    Vector {
        handler: UsageFault,
    },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { handler: SVCall },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { handler: PendSV },
    Vector { handler: SysTick },
];
```

Some of the entries in the vector table are *reserved*; the ARM documentation states that they should be assigned the value `0` so we use a union to do exactly that. The entries that must point to a handler make use of *external* functions; this is important because it lets the end user *provide* the actual function definition.

Next, we define a default exception handler in the Rust code. Exceptions that have not been assigned a handler by the end user will make use of this default handler.

```
$ tail -n4 ../rt/src/lib.rs
```

```
#[no_mangle]
pub extern "C" fn DefaultExceptionHandler() {
    loop {}
}
```

Linker script side

On the linker script side, we place these new exception vectors right after the reset vector.

```
$ sed -n 12,25p ../rt/link.x
```

```
EXTERN(RESET_VECTOR);
EXTERN(EXCEPTIONS); /* <- NEW */

SECTIONS
{
  .vector_table ORIGIN(FLASH) :
  {
    /* First entry: initial Stack Pointer value */
    LONG(ORIGIN(RAM) + LENGTH(RAM));

    /* Second entry: reset vector */
    KEEP(*(.vector_table.reset_vector));

    /* The next 14 entries are exception vectors */
    KEEP(*(.vector_table.exceptions)); /* <- NEW */
  } > FLASH
```

And we use `PROVIDE` to give a default value to the handlers that we left undefined in `rt` (`NMI` and the others above):

```
$ tail -n8 ../rt/link.x
```

```
PROVIDE(NMI = DefaultExceptionHandler);
PROVIDE(HardFault = DefaultExceptionHandler);
PROVIDE(MemManage = DefaultExceptionHandler);
PROVIDE(BusFault = DefaultExceptionHandler);
PROVIDE(UsageFault = DefaultExceptionHandler);
PROVIDE(SVCall = DefaultExceptionHandler);
PROVIDE(PendSV = DefaultExceptionHandler);
PROVIDE(SysTick = DefaultExceptionHandler);
```

`PROVIDE` only takes effect when the symbol to the left of the equal sign is still undefined after inspecting all the input object files. This is the scenario where the user didn't implement the handler for the respective exception.

Testing it

That's it! The `rt` crate now has support for exception handlers. We can test it out with following application:

> **NOTE**: Turns out it's hard to generate an exception in QEMU. On real hardware a read to an invalid memory address (i.e. outside of the Flash and RAM regions) would be enough but QEMU happily accepts the operation and returns zero. A trap instruction works on both QEMU and hardware but unfortunately it's not available on stable so you'll have to temporarily switch to nightly to run this and the next example.

```
#![feature(core_intrinsics)]
#![no_main]
#![no_std]

use core::intrinsics;

use rt::entry;

entry!(main);

fn main() -> ! {
    // this executes the undefined instruction (UDF) and causes a HardFault
exception
    intrinsics::abort()
}
```

```
(gdb) target remote :3333
Remote debugging using :3333
Reset () at ../rt/src/lib.rs:7
7       pub unsafe extern "C" fn Reset() -> ! {

(gdb) b DefaultExceptionHandler
Breakpoint 1 at 0xec: file ../rt/src/lib.rs, line 95.

(gdb) continue
Continuing.

Breakpoint 1, DefaultExceptionHandler ()
    at ../rt/src/lib.rs:95
95          loop {}

(gdb) list
90          Vector { handler: SysTick },
91      ];
92
93      #[no_mangle]
94      pub extern "C" fn DefaultExceptionHandler() {
95          loop {}
96      }
```

And for completeness, here's the disassembly of the optimized version of the program:

```
$ cargo objdump --bin app --release -- -d --no-show-raw-insn --print-imm-hex
```

```
app:     file format elf32-littlearm

Disassembly of section .text:

<main>:
                trap
                trap

<Reset>:
                push    {r7, lr}
                mov     r7, sp
                movw    r1, #0x0
                movw    r0, #0x0
                movt    r1, #0x2000
                movt    r0, #0x2000
                subs    r1, r1, r0
                bl      0x9c <__aeabi_memclr>   @ imm = #0x3e
                movw    r1, #0x0
                movw    r0, #0x0
                movt    r1, #0x2000
                movt    r0, #0x2000
                subs    r2, r1, r0
                movw    r1, #0x282
                movt    r1, #0x0
                bl      0x84 <__aeabi_memcpy>   @ imm = #0x8
                bl      0x40 <main>             @ imm = #-0x40
                trap

<UsageFault>:
```

```
$ cargo objdump --bin app --release -- -s -j .vector_table
```

```
app:     file format elf32-littlearm
Contents of section .vector_table:
 0000 00000120 45000000 83000000 83000000  ... E...........
 0010 83000000 83000000 83000000 00000000  ................
 0020 00000000 00000000 00000000 83000000  ................
 0030 00000000 00000000 83000000 83000000  ................
```

The vector table now resembles the results of all the code snippets in this book so far. To summarize:

- In the *Inspecting it* section of the earlier memory chapter, we learned that:
  - The first entry in the vector table contains the initial value of the stack pointer.
  - Objdump prints in `little endian` format, so the stack starts at `0x2001_0000`.
  - The second entry points to address `0x0000_0045`, the Reset handler.
    - The address of the Reset handler can be seen in the disassembly above, being `0x44`.
    - The first bit being set to 1 does not alter the address due to alignment

requirements. Instead, it causes the function to be executed in *thumb mode*.
- Afterwards, a pattern of addresses alternating between `0x83` and `0x00` is visible.
  - Looking at the disassembly above, it is clear that `0x83` refers to the `DefaultExceptionHandler` (`0x84` executed in thumb mode).
  - Cross referencing the pattern to the vector table that was set up earlier in this chapter (see the definition of `pub static EXCEPTIONS`) with the vector table layout for the Cortex-M, it is clear that the address of the `DefaultExceptionHandler` is present each time a respective handler entry is present in the table.
  - In turn, it is also visible that the layout of the vector table data structure in the Rust code is aligned with all the reserved slots in the Cortex-M vector table. Hence, all reserved slots are correctly set to a value of zero.

Overriding a handler

To override an exception handler, the user has to provide a function whose symbol name exactly matches the name we used in `EXCEPTIONS` .

```rust
#![feature(core_intrinsics)]
#![no_main]
#![no_std]

use core::intrinsics;

use rt::entry;

entry!(main);

fn main() -> ! {
    intrinsics::abort()
}

#[no_mangle]
pub extern "C" fn HardFault() -> ! {
    // do something interesting here
    loop {}
}
```

You can test it in QEMU

```
(gdb) target remote :3333
Remote debugging using :3333
Reset () at /home/japaric/rust/embedonomicon/ci/exceptions/rt/src/lib.rs:7
7       pub unsafe extern "C" fn Reset() -> ! {

(gdb) b HardFault
Breakpoint 1 at 0x44: file src/main.rs, line 18.

(gdb) continue
Continuing.

Breakpoint 1, HardFault () at src/main.rs:18
18          loop {}

(gdb) list
13      }
14
15      #[no_mangle]
16      pub extern "C" fn HardFault() -> ! {
17          // do something interesting here
18          loop {}
19      }
```

The program now executes the user defined `HardFault` function instead of the `DefaultExceptionHandler` in the `rt` crate.

Like our first attempt at a `main` interface, this first implementation has the problem of having no type safety. It's also easy to mistype the name of the exception, but that doesn't produce an error or warning. Instead the user defined handler is simply ignored. Those problems can be fixed using a macro like the `exception!` macro defined in `cortex-m-rt` v0.5.x or the `exception` attribute in `cortex-m-rt` v0.6.x.

Assembly on stable

> Note: Since Rust 1.59, both *inline* assembly (`asm!`) and *free form* assembly (`global_asm!`) become stable. But since it will take some time for the existing crates to catchup the change, and since it's good for us to know the other ways in history we used to deal with assembly, we will keep this chapter here.

So far we have managed to boot the device and handle interrupts without a single line of assembly. That's quite a feat! But depending on the architecture you are targeting you may need some assembly to get to this point. There are also some operations like context switching that require assembly, etc.

The problem is that both *inline* assembly (`asm!`) and *free form* assembly (`global_asm!`) are unstable, and there's no estimate for when they'll be stabilized, so you can't use them on stable . This is not a showstopper because there are some workarounds which we'll document here.

To motivate this section we'll tweak the `HardFault` handler to provide information about the stack frame that generated the exception.

Here's what we want to do:

Instead of letting the user directly put their `HardFault` handler in the vector table we'll make the `rt` crate put a trampoline to the user-defined `HardFault` handler in the vector table.

```
$ tail -n36 ../rt/src/lib.rs
```

```
extern "C" {
    fn NMI();
    fn HardFaultTrampoline(); // <- CHANGED!
    fn MemManage();
    fn BusFault();
    fn UsageFault();
    fn SVCall();
    fn PendSV();
    fn SysTick();
}

#[link_section = ".vector_table.exceptions"]
#[no_mangle]
pub static EXCEPTIONS: [Vector; 14] = [
    Vector { handler: NMI },
    Vector { handler: HardFaultTrampoline }, // <- CHANGED!
    Vector { handler: MemManage },
    Vector { handler: BusFault },
    Vector {
        handler: UsageFault,
    },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { handler: SVCall },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { handler: PendSV },
    Vector { handler: SysTick },
];

#[no_mangle]
pub extern "C" fn DefaultExceptionHandler() {
    loop {}
}
```

This trampoline will read the stack pointer and then call the user `HardFault` handler. The trampoline will have to be written in assembly:

```
  mrs r0, MSP
  b HardFault
```

Due to how the ARM ABI works this sets the Main Stack Pointer (MSP) as the first argument of the `HardFault` function / routine. This MSP value also happens to be a pointer to the registers pushed to the stack by the exception. With these changes the user `HardFault` handler must now have signature `fn(&StackedRegisters) -> !`.

`.s` files

One approach to stable assembly is to write the assembly in an external file:

```
$ cat ../rt/asm.s
```

```
  .section .text.HardFaultTrampoline
  .global HardFaultTrampoline
  .thumb_func
HardFaultTrampoline:
  mrs r0, MSP
  b HardFault
```

And use the `cc` crate in the build script of the `rt` crate to assemble that file into an object file (`.o`) and then into an archive (`.a`).

```
$ cat ../rt/build.rs
```

```
use std::{env, error::Error, fs::File, io::Write, path::PathBuf};

use cc::Build;

fn main() -> Result<(), Box<dyn Error>> {
    // build directory for this crate
    let out_dir = PathBuf::from(env::var_os("OUT_DIR").unwrap());

    // extend the library search path
    println!("cargo:rustc-link-search={}", out_dir.display());

    // put `link.x` in the build directory
    File::create(out_dir.join("link.x"))?.write_all(include_bytes!
("link.x"))?;

    // assemble the `asm.s` file
    Build::new().file("asm.s").compile("asm"); // <- NEW!

    // rebuild if `asm.s` changed
    println!("cargo:rerun-if-changed=asm.s"); // <- NEW!

    Ok(())
}
```

```
$ tail -n2 ../rt/Cargo.toml
```

```
[build-dependencies]
cc = "1.0.25"
```

And that's it!

We can confirm that the vector table contains a pointer to `HardFaultTrampoline` by writing a very simple program.

```
#![no_main]
#![no_std]

use rt::entry;

entry!(main);

fn main() -> ! {
    loop {}
}

#[allow(non_snake_case)]
#[no_mangle]
pub fn HardFault(_ef: *const u32) -> ! {
    loop {}
}
```

Here's the disassembly. Look at the address of `HardFaultTrampoline` .

```
$ cargo objdump --bin app --release -- -d --no-show-raw-insn --print-imm-hex
```

```


app:    file format elf32-littlearm

Disassembly of section .text:

<HardFault>:
                b       0x40 <HardFault>        @ imm = #-0x4

<main>:
                b       0x42 <main>             @ imm = #-0x4

<Reset>:
                push    {r7, lr}
                mov     r7, sp
                bl      0x42 <main>             @ imm = #-0xa
                trap

<UsageFault>:
                b       0x4e <UsageFault>       @ imm = #-0x4

<HardFaultTrampoline>:
                mrs     r0, msp
                b       0x40 <HardFault>        @ imm = #-0x18
```

NOTE: To make this disassembly smaller I commented out the initialization of RAM

Now look at the vector table. The 4th entry should be the address of `HardFaultTrampoline` plus one.

```
$ cargo objdump --bin app --release -- -s -j .vector_table
```

```
app:    file format elf32-littlearm
Contents of section .vector_table:
 0000 00000120 45000000 4f000000 51000000  ... E...O...Q...
 0010 4f000000 4f000000 4f000000 00000000  O...O...O.......
 0020 00000000 00000000 00000000 4f000000  ............O...
 0030 00000000 00000000 4f000000 4f000000  ........O...O...
```

.o / .a files

The downside of using the `cc` crate is that it requires some assembler program on the build machine. For example when targeting ARM Cortex-M the `cc` crate uses `arm-none-eabi-gcc` as the assembler.

Instead of assembling the file on the build machine we can ship a pre-assembled file with the `rt` crate. That way no assembler program is required on the build machine. However, you would still need an assembler on the machine that packages and publishes the crate.

There's not much difference between an assembly (`.s`) file and its *compiled* version: the object (`.o`) file. The assembler doesn't do any optimization; it simply chooses the right object file format for the target architecture.

Cargo provides support for bundling archives (`.a`) with crates. We can package object files into an archive using the `ar` command and then bundle the archive with the crate. In fact, this what the `cc` crate does; you can see the commands it invoked by searching for a file named `output` in the `target` directory.

```
$ grep running $(find target -name output)
```

```
running: "arm-none-eabi-gcc" "-O0" "-ffunction-sections" "-fdata-sections"
"-fPIC" "-g" "-fno-omit-frame-pointer" "-mthumb" "-march=armv7-m" "-Wall"
"-Wextra" "-o" "/tmp/app/target/thumbv7m-none-eabi/debug/build
/rt-6ee84e54724f2044/out/asm.o" "-c" "asm.s"
running: "ar" "crs" "/tmp/app/target/thumbv7m-none-eabi/debug/build
/rt-6ee84e54724f2044/out/libasm.a" "/home/japaric/rust-embedded/embedonomicon
/ci/asm/app/target/thumbv7m-none-eabi/debug/build/rt-6ee84e54724f2044
/out/asm.o"
```

```
$ grep cargo $(find target -name output)
```

```
cargo:rustc-link-search=/tmp/app/target/thumbv7m-none-eabi/debug/build
/rt-6ee84e54724f2044/out
cargo:rustc-link-lib=static=asm
cargo:rustc-link-search=native=/tmp/app/target/thumbv7m-none-eabi/debug/build
/rt-6ee84e54724f2044/out
```

We'll do something similar to produce an archive.

```
$ # most of flags `cc` uses have no effect when assembling so we drop them
$ arm-none-eabi-as -march=armv7-m asm.s -o asm.o

$ ar crs librt.a asm.o

$ arm-none-eabi-objdump -Cd librt.a


In archive librt.a:

asm.o:     file format elf32-littlearm


Disassembly of section .text.HardFaultTrampoline:

00000000 <HardFaultTrampoline>:
   0:   f3ef 8008       mrs     r0, MSP
   4:   e7fe            b.n     0 <HardFault>
```

Next we modify the build script to bundle this archive with the `rt` rlib.

```
$ cat ../rt/build.rs
```

```
use std::{
    env,
    error::Error,
    fs::{self, File},
    io::Write,
    path::PathBuf,
};

fn main() -> Result<(), Box<dyn Error>> {
    // build directory for this crate
    let out_dir = PathBuf::from(env::var_os("OUT_DIR").unwrap());

    // extend the library search path
    println!("cargo:rustc-link-search={}", out_dir.display());

    // put `link.x` in the build directory
    File::create(out_dir.join("link.x"))?.write_all(include_bytes!
("link.x"))?;

    // link to `librt.a`
    fs::copy("librt.a", out_dir.join("librt.a"))?; // <- NEW!
    println!("cargo:rustc-link-lib=static=rt"); // <- NEW!

    // rebuild if `librt.a` changed
    println!("cargo:rerun-if-changed=librt.a"); // <- NEW!

    Ok(())
}
```

Now we can test this new version against the simple program from before and we'll get the same output.

```
$ cargo objdump --bin app --release -- -d --no-show-raw-insn --print-imm-hex
```

```
app:    file format elf32-littlearm

Disassembly of section .text:

<HardFault>:
                b       0x40 <HardFault>        @ imm = #-0x4

<main>:
                b       0x42 <main>             @ imm = #-0x4

<Reset>:
                push    {r7, lr}
                mov     r7, sp
                bl      0x42 <main>             @ imm = #-0xa
                trap

<UsageFault>:
                b       0x4e <UsageFault>       @ imm = #-0x4

<HardFaultTrampoline>:
                mrs     r0, msp
                b       0x40 <HardFault>        @ imm = #-0x18
```

NOTE: As before I have commented out the RAM initialization to make the disassembly smaller.

```
$ cargo objdump --bin app --release -- -s -j .vector_table
```

```
app:    file format elf32-littlearm
Contents of section .vector_table:
 0000 00000120 45000000 4f000000 51000000  ... E...O...Q...
 0010 4f000000 4f000000 4f000000 00000000  O...O...O.......
 0020 00000000 00000000 00000000 4f000000  ............O...
 0030 00000000 00000000 4f000000 4f000000  ........O...O...
```

The downside of shipping pre-assembled archives is that, in the worst case scenario, you'll need to ship one build artifact for each compilation target your library supports.

Logging with symbols

This section will show you how to utilize symbols and the ELF format to achieve super cheap logging.

Arbitrary symbols

Whenever we needed a stable symbol interface between crates we have mainly used the `no_mangle` attribute and sometimes the `export_name` attribute. The `export_name` attribute takes a string which becomes the name of the symbol whereas `#[no_mangle]` is basically sugar for `#[export_name = <item-name>]`.

Turns out we are not limited to single word names; we can use arbitrary strings, e.g. sentences, as the argument of the `export_name` attribute. As least when the output format is ELF anything that doesn't contain a null byte is fine.

Let's check that out:

```
$ cargo new --lib foo

$ cat foo/src/lib.rs
```

```
#[export_name = "Hello, world!"]
#[used]
static A: u8 = 0;

#[export_name = "こんにちは"]
#[used]
static B: u8 = 0;
```

```
$ ( cd foo && cargo nm --lib )
foo-d26a39c34b4e80ce.3lnzqy0jbpxj4pld.rcgu.o:
0000000000000000 r Hello, world!
0000000000000000 V __rustc_debug_gdb_scripts_section__
0000000000000000 r こんにちは
```

Can you see where this is going?

Encoding

Here's what we'll do: we'll create one `static` variable per log message but instead of storing the messages *in* the variables we'll store the messages in the variables' *symbol names*. What we'll log then will not be the contents of the `static` variables but their addresses.

As long as the `static` variables are not zero sized each one will have a different address. What we're doing here is effectively encoding each message into a unique identifier, which happens to be the variable address. Some part of the log system will have to decode this id back into the message.

Let's write some code to illustrate the idea.

In this example we'll need some way to do I/O so we'll use the `cortex-m-semihosting` crate for that. Semihosting is a technique for having a target device borrow the host I/O capabilities; the host here usually refers to the machine that's debugging the target device. In our case, QEMU supports semihosting out of the box so there's no need for a debugger. On a real device you'll have other ways to do I/O like a serial port; we use semihosting in this case because it's the easiest way to do I/O on QEMU.

Here's the code

```
#![no_main]
#![no_std]

use core::fmt::Write;
use cortex_m_semihosting::{debug, hio};

use rt::entry;

entry!(main);

fn main() -> ! {
    let mut hstdout = hio::hstdout().unwrap();

    #[export_name = "Hello, world!"]
    static A: u8 = 0;

    let _ = writeln!(hstdout, "{:#x}", &A as *const u8 as usize);

    #[export_name = "Goodbye"]
    static B: u8 = 0;

    let _ = writeln!(hstdout, "{:#x}", &B as *const u8 as usize);

    debug::exit(debug::EXIT_SUCCESS);

    loop {}
}
```

We also make use of the `debug::exit` API to have the program terminate the QEMU process. This is a convenience so we don't have to manually terminate the QEMU process.

And here's the `dependencies` section of the Cargo.toml:

```
[dependencies]
cortex-m-semihosting = "0.3.1"
rt = { path = "../rt" }
```

Now we can build the program

```
$ cargo build
```

To run it we'll have to add the `--semihosting-config` flag to our QEMU invocation:

```
$ qemu-system-arm \
      -cpu cortex-m3 \
      -machine lm3s6965evb \
      -nographic \
      -semihosting-config enable=on,target=native \
      -kernel target/thumbv7m-none-eabi/debug/app
```

```
0x1fe0
0x1fe1
```

> **NOTE**: These addresses may not be the ones you get locally because addresses of `static` variable are not guaranteed to remain the same when the toolchain is changed (e.g. optimizations may have improved).

Now we have two addresses printed to the console.

Decoding

How do we convert these addresses into strings? The answer is in the symbol table of the ELF file.

```
$ cargo objdump --bin app -- -t | grep '\.rodata\s*0*1\b'
```

```
00001fe1 g       .rodata                00000001 Goodbye
00001fe0 g       .rodata                00000001 Hello, world!
$ # first column is the symbol address; last column is the symbol name
```

`objdump -t` prints the symbol table. This table contains *all* the symbols but we are only looking for the ones in the `.rodata` section and whose size is one byte (our variables have type `u8`).

It's important to note that the address of the symbols will likely change when optimizing the program. Let's check that.

> **PROTIP** You can set `target.thumbv7m-none-eabi.runner` to the long QEMU command from before (`qemu-system-arm -cpu (..) -kernel`) in the Cargo configuration file (`.cargo/conifg`) to have `cargo run` use that *runner* to execute the output binary.

```
$ head -n2 .cargo/config
```

```
[target.thumbv7m-none-eabi]
runner = "qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic
-semihosting-config enable=on,target=native -kernel"
```

```
$ cargo run --release
     Running `qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic
-semihosting-config enable=on,target=native -kernel target/thumbv7m-none-
eabi/release/app`
```

```
0xb9c
0xb9d
```

```
$ cargo objdump --bin app --release -- -t | grep '\.rodata\s*0*1\b'
```

```
00000b9d g     O .rodata        00000001 Goodbye
00000b9c g     O .rodata        00000001 Hello, world!
```

So make sure to always look for the strings in the ELF file you executed.

Of course, the process of looking up the strings in the ELF file can be automated using a tool that parses the symbol table (`.symtab` section) contained in the ELF file. Implementing such tool is out of scope for this book and it's left as an exercise for the reader.

Making it zero cost

Can we do better? Yes, we can!

The current implementation places the `static` variables in `.rodata` , which means they occupy size in Flash even though we never use their contents. Using a little bit of linker script magic we can make them occupy *zero* space in Flash.

```
$ cat log.x
```

```
SECTIONS
{
  .log 0 (INFO) : {
    *(.log);
  }
}
```

We'll place the `static` variables in this new output `.log` section. This linker script will collect all the symbols in the `.log` sections of input object files and put them in an output `.log` section. We have seen this pattern in the Memory layout chapter.

The new bit here is the `(INFO)` part; this tells the linker that this section is a non-allocatable section. Non-allocatable sections are kept in the ELF binary as metadata but they are not loaded onto the target device.

We also specified the start address of this output section: the `0` in `.log 0 (INFO)`.

The other improvement we can do is switch from formatted I/O (`fmt::Write`) to binary I/O, that is send the addresses to the host as bytes rather than as strings.

Binary serialization can be hard but we'll keep things super simple by serializing each address as a single byte. With this approach we don't have to worry about endianness or framing. The downside of this format is that a single byte can only represent up to 256 different addresses.

Let's make those changes:

```
#![no_main]
#![no_std]

use cortex_m_semihosting::{debug, hio};

use rt::entry;

entry!(main);

fn main() -> ! {
    let mut hstdout = hio::hstdout().unwrap();

    #[export_name = "Hello, world!"]
    #[link_section = ".log"] // <- NEW!
    static A: u8 = 0;

    let address = &A as *const u8 as usize as u8;
    hstdout.write_all(&[address]).unwrap(); // <- CHANGED!

    #[export_name = "Goodbye"]
    #[link_section = ".log"] // <- NEW!
    static B: u8 = 0;

    let address = &B as *const u8 as usize as u8;
    hstdout.write_all(&[address]).unwrap(); // <- CHANGED!

    debug::exit(debug::EXIT_SUCCESS);

    loop {}
}
```

Before you run this you'll have to append `-Tlog.x` to the arguments passed to the linker. That can be done in the Cargo configuration file.

```
$ cat .cargo/config
```

```
[target.thumbv7m-none-eabi]
runner = "qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic
-semihosting-config enable=on,target=native -kernel"
rustflags = [
  "-C", "link-arg=-Tlink.x",
  "-C", "link-arg=-Tlog.x", # <- NEW!
]

[build]
target = "thumbv7m-none-eabi"
```

Now you can run it! Since the output now has a binary format we'll pipe it through the `xxd` command to reformat it as a hexadecimal string.

```
$ cargo run | xxd -p
```

```
0001
```

The addresses are `0x00` and `0x01`. Let's now look at the symbol table.

```
$ cargo objdump --bin app -- -t | grep '\.log'
```

```
00000001 g     O .log   00000001 Goodbye
00000000 g     O .log   00000001 Hello, world!
```

There are our strings. You'll notice that their addresses now start at zero; this is because we set a start address for the output `.log` section.

Each variable is 1 byte in size because we are using `u8` as their type. If we used something like `u16` then all address would be even and we would not be able to efficiently use all the address space (`0...255`).

Packaging it up

You've noticed that the steps to log a string are always the same so we can refactor them into a macro that lives in its own crate. Also, we can make the logging library more reusable by abstracting the I/O part behind a trait.

```
$ cargo new --lib log

$ cat log/src/lib.rs
```

```
#![no_std]

pub trait Log {
    type Error;

    fn log(&mut self, address: u8) -> Result<(), Self::Error>;
}

#[macro_export]
macro_rules! log {
    ($logger:expr, $string:expr) => {{
        #[export_name = $string]
        #[link_section = ".log"]
        static SYMBOL: u8 = 0;

        $crate::Log::log(&mut $logger, &SYMBOL as *const u8 as usize as u8)
    }};
}
```

Given that this library depends on the `.log` section it should be its responsibility to provide the `log.x` linker script so let's make that happen.

```
$ mv log.x ../log/
```

```
$ cat ../log/build.rs
```

```rust
use std::{env, error::Error, fs::File, io::Write, path::PathBuf};

fn main() -> Result<(), Box<dyn Error>> {
    // Put the linker script somewhere the linker can find it
    let out = PathBuf::from(env::var("OUT_DIR")?);

    File::create(out.join("log.x"))?.write_all(include_bytes!("log.x"))?;

    println!("cargo:rustc-link-search={}", out.display());

    Ok(())
}
```

Now we can refactor our application to use the `log!` macro:

```
$ cat src/main.rs
```

```rust
#![no_main]
#![no_std]

use cortex_m_semihosting::{
    debug,
    hio::{self, HStdout},
};

use log::{log, Log};
use rt::entry;

struct Logger {
    hstdout: HStdout,
}

impl Log for Logger {
    type Error = ();

    fn log(&mut self, address: u8) -> Result<(), ()> {
        self.hstdout.write_all(&[address])
    }
}

entry!(main);

fn main() -> ! {
    let hstdout = hio::hstdout().unwrap();
    let mut logger = Logger { hstdout };

    let _ = log!(logger, "Hello, world!");

    let _ = log!(logger, "Goodbye");

    debug::exit(debug::EXIT_SUCCESS);

    loop {}
}
```

Don't forget to update the `Cargo.toml` file to depend on the new `log` crate.

```console
$ tail -n4 Cargo.toml
```

```toml
[dependencies]
cortex-m-semihosting = "0.3.1"
log = { path = "../log" }
rt = { path = "../rt" }
```

```console
$ cargo run | xxd -p
```

```text
0001
```

```
$ cargo objdump --bin app -- -t | grep '\.log'
```

```
00000001 g     O .log   00000001 Goodbye
00000000 g     O .log   00000001 Hello, world!
```

Same output as before!

Bonus: Multiple log levels

Many logging frameworks provide ways to log messages at different *log levels*. These log levels convey the severity of the message: "this is an error", "this is just a warning", etc. These log levels can be used to filter out unimportant messages when searching for e.g. error messages.

We can extend our logging library to support log levels without increasing its footprint. Here's how we'll do that:

We have a flat address space for the messages: from `0` to `255` (inclusive). To keep things simple let's say we only want to differentiate between error messages and warning messages. We can place all the error messages at the beginning of the address space, and all the warning messages *after* the error messages. If the decoder knows the address of the first warning message then it can classify the messages. This idea can be extended to support more than two log levels.

Let's test the idea by replacing the `log` macro with two new macros: `error!` and `warn!`.

```
$ cat ../log/src/lib.rs
```

```
#![no_std]

pub trait Log {
    type Error;

    fn log(&mut self, address: u8) -> Result<(), Self::Error>;
}

/// Logs messages at the ERROR log level
#[macro_export]
macro_rules! error {
    ($logger:expr, $string:expr) => {{
        #[export_name = $string]
        #[link_section = ".log.error"] // <- CHANGED!
        static SYMBOL: u8 = 0;

        $crate::Log::log(&mut $logger, &SYMBOL as *const u8 as usize as u8)
    }};
}

/// Logs messages at the WARNING log level
#[macro_export]
macro_rules! warn {
    ($logger:expr, $string:expr) => {{
        #[export_name = $string]
        #[link_section = ".log.warning"] // <- CHANGED!
        static SYMBOL: u8 = 0;

        $crate::Log::log(&mut $logger, &SYMBOL as *const u8 as usize as u8)
    }};
}
```

We distinguish errors from warnings by placing the messages in different link sections.

The next thing we have to do is update the linker script to place error messages before the warning messages.

```
$ cat ../log/log.x
```

```
SECTIONS
{
  .log 0 (INFO) : {
    *(.log.error);
    __log_warning_start__ = .;
    *(.log.warning);
  }
}
```

We also give a name, `__log_warning_start__`, to the boundary between the errors and the warnings. The address of this symbol will be the address of the first warning message.

We can now update the application to make use of these new macros.

```
$ cat src/main.rs
```

```rust
#![no_main]
#![no_std]

use cortex_m_semihosting::{
    debug,
    hio::{self, HStdout},
};

use log::{error, warn, Log};
use rt::entry;

entry!(main);

fn main() -> ! {
    let hstdout = hio::hstdout().unwrap();
    let mut logger = Logger { hstdout };

    let _ = warn!(logger, "Hello, world!"); // <- CHANGED!

    let _ = error!(logger, "Goodbye"); // <- CHANGED!

    debug::exit(debug::EXIT_SUCCESS);

    loop {}
}

struct Logger {
    hstdout: HStdout,
}

impl Log for Logger {
    type Error = ();

    fn log(&mut self, address: u8) -> Result<(), ()> {
        self.hstdout.write_all(&[address])
    }
}
```

The output won't change much:

```
$ cargo run | xxd -p
```

```
0100
```

We still get two bytes in the output but the error is given the address 0 and the warning is given the address 1 even though the warning was logged first.

Now look at the symbol table.

```
$ cargo objdump --bin app -- -t | grep '\.log'
```

```
00000000 g     O .log   00000001 Goodbye
00000001 g     O .log   00000001 Hello, world!
00000001 g       .log   00000000 __log_warning_start__
```

There's now an extra symbol, `__log_warning_start__`, in the `.log` section. The address of this symbol is the address of the first warning message. Symbols with addresses lower than this value are errors, and the rest of symbols are warnings.

With an appropriate decoder you could get the following human readable output from all this information:

```
WARNING Hello, world!
ERROR Goodbye
```

If you liked this section check out the `stlog` logging framework which is a complete implementation of this idea.

Global singletons

In this section we'll cover how to implement a global, shared singleton. The embedded Rust book covered local, owned singletons which are pretty much unique to Rust. Global singletons are essentially the singleton pattern you see in C and C++; they are not specific to embedded development but since they involve symbols they seemed a good fit for the embedonomicon.

> **TODO**(resources team) link "the embedded Rust book" to the singletons section when it's up

To illustrate this section we'll extend the logger we developed in the last section to support global logging. The result will be very similar to the `#[global_allocator]` feature covered in the embedded Rust book.

> **TODO**(resources team) link `#[global_allocator]` to the collections chapter of the book when it's in a more stable location.

Here's the summary of what we want to:

In the last section we created a `log!` macro to log messages through a specific logger, a value that implements the `Log` trait. The syntax of the `log!` macro is `log!(logger, "String")`. We want to extend the macro such that `log!("String")` also works. Using the `logger` -less version should log the message through a global logger; this is how `std::println!` works. We'll also need a mechanism to declare what the global logger is; this is the part that's similar to `#[global_allocator]`.

It could be that the global logger is declared in the top crate and it could also be that the type of the global logger is defined in the top crate. In this scenario the dependencies can *not* know the exact type of the global logger. To support this scenario we'll need some indirection.

Instead of hardcoding the type of the global logger in the `log` crate we'll declare only the *interface* of the global logger in that crate. That is we'll add a new trait, `GlobalLog`, to the `log` crate. The `log!` macro will also have to make use of that trait.

```
$ cat ../log/src/lib.rs
```

```
#![no_std]

// NEW!
pub trait GlobalLog: Sync {
    fn log(&self, address: u8);
}

pub trait Log {
    type Error;

    fn log(&mut self, address: u8) -> Result<(), Self::Error>;
}

#[macro_export]
macro_rules! log {
    // NEW!
    ($string:expr) => {
        unsafe {
            extern "Rust" {
                static LOGGER: &'static dyn $crate::GlobalLog;
            }

            #[export_name = $string]
            #[link_section = ".log"]
            static SYMBOL: u8 = 0;

            $crate::GlobalLog::log(LOGGER, &SYMBOL as *const u8 as usize as
u8)
        }
    };

    ($logger:expr, $string:expr) => {{
        #[export_name = $string]
        #[link_section = ".log"]
        static SYMBOL: u8 = 0;

        $crate::Log::log(&mut $logger, &SYMBOL as *const u8 as usize as u8)
    }};
}

// NEW!
#[macro_export]
macro_rules! global_logger {
    ($logger:expr) => {
        #[no_mangle]
        pub static LOGGER: &dyn $crate::GlobalLog = &$logger;
    };
}
```

There's quite a bit to unpack here.

Let's start with the trait.

```
pub trait GlobalLog: Sync {
    fn log(&self, address: u8);
}
```

Both `GlobalLog` and `Log` have a `log` method. The difference is that `GlobalLog.log` takes a shared reference to the receiver (`&self`). This is necessary because the global logger will be a `static` variable. More on that later.

The other difference is that `GlobalLog.log` doesn't return a `Result` . This means that it can *not* report errors to the caller. This is not a strict requirement for traits used to implement global singletons. Error handling in global singletons is fine but then all users of the global version of the `log!` macro have to agree on the error type. Here we are simplifying the interface a bit by having the `GlobalLog` implementer deal with the errors.

Yet another difference is that `GlobalLog` requires that the implementer is `Sync` , that is that it can be shared between threads. This is a requirement for values placed in `static` variables; their types must implement the `Sync` trait.

At this point it may not be entirely clear why the interface has to look this way. The other parts of the crate will make this clearer so keep reading.

Next up is the `log!` macro:

```
    ($string:expr) => {
        unsafe {
            extern "Rust" {
                static LOGGER: &'static dyn $crate::GlobalLog;
            }

            #[export_name = $string]
            #[link_section = ".log"]
            static SYMBOL: u8 = 0;

            $crate::GlobalLog::log(LOGGER, &SYMBOL as *const u8 as usize as
u8)
        }
    };
```

When called without a specific `$logger` the macros uses an `extern` `static` variable called `LOGGER` to log the message. This variable *is* the global logger that's defined somewhere else; that's why we use the `extern` block. We saw this pattern in the main interface chapter.

We need to declare a type for `LOGGER` or the code won't type check. We don't know the concrete type of `LOGGER` at this point but we know, or rather require, that it implements the `GlobalLog` trait so we can use a trait object here.

The rest of the macro expansion looks very similar to the expansion of the local version of the `log!` macro so I won't explain it here as it's explained in the previous chapter.

Now that we know that `LOGGER` has to be a trait object it's clearer why we omitted the associated `Error` type in `GlobalLog`. If we had not omitted then we would have need to pick a type for `Error` in the type signature of `LOGGER`. This is what I earlier meant by "all users of `log!` would need to agree on the error type".

Now the final piece: the `global_logger!` macro. It could have been a proc macro attribute but it's easier to write a `macro_rules!` macro.

```
#[macro_export]
macro_rules! global_logger {
    ($logger:expr) => {
        #[no_mangle]
        pub static LOGGER: &dyn $crate::GlobalLog = &$logger;
    };
}
```

This macro creates the `LOGGER` variable that `log!` uses. Because we need a stable ABI interface we use the `no_mangle` attribute. This way the symbol name of `LOGGER` will be "LOGGER" which is what the `log!` macro expects.

The other important bit is that the type of this static variable must exactly match the type used in the expansion of the `log!` macro. If they don't match Bad Stuff will happen due to ABI mismatch.

Let's write an example that uses this new global logger functionality.

```
$ cat src/main.rs
```

```
#![no_main]
#![no_std]

use cortex_m::interrupt;
use cortex_m_semihosting::{
    debug,
    hio::{self, HStdout},
};

use log::{global_logger, log, GlobalLog};
use rt::entry;

struct Logger;

global_logger!(Logger);

entry!(main);

fn main() -> ! {
    log!("Hello, world!");

    log!("Goodbye");

    debug::exit(debug::EXIT_SUCCESS);

    loop {}
}

impl GlobalLog for Logger {
    fn log(&self, address: u8) {
        // we use a critical section (`interrupt::free`) to make the access
to the
        // `static mut` variable interrupt safe which is required for memory
safety
        interrupt::free(|_| unsafe {
            static mut HSTDOUT: Option<HStdout> = None;

            // lazy initialization
            if HSTDOUT.is_none() {
                HSTDOUT = Some(hio::hstdout()?);
            }

            let hstdout = HSTDOUT.as_mut().unwrap();

            hstdout.write_all(&[address])
        }).ok(); // `.ok()` = ignore errors
    }
}
```

TODO(resources team) use `cortex_m::Mutex` instead of a `static mut` variable when `const fn` is stabilized.

We had to add `cortex-m` to the dependencies.

```
$ tail -n5 Cargo.toml
```

```
[dependencies]
cortex-m = "0.5.7"
cortex-m-semihosting = "0.3.1"
log = { path = "../log" }
rt = { path = "../rt" }
```

This is a port of one of the examples written in the previous section. The output is the same as what we got back there.

```
$ cargo run | xxd -p
```

```
0001
```

```
$ cargo objdump --bin app -- -t | grep '\.log'
```

```
00000001 g     O .log   00000001 Goodbye
00000000 g     O .log   00000001 Hello, world!
```

Some readers may be concerned about this implementation of global singletons not being zero cost because it uses trait objects which involve dynamic dispatch, that is method calls are performed through a vtable lookup.

However, it appears that LLVM is smart enough to eliminate the dynamic dispatch when compiling with optimizations / LTO. This can be confirmed by searching for `LOGGER` in the symbol table.

```
$ cargo objdump --bin app --release -- -t | grep LOGGER
```

If the `static` is missing that means that there is no vtable and that LLVM was capable of transforming all the `LOGGER.log` calls into `Logger.log` calls.

Direct Memory Access (DMA)

This section covers the core requirements for building a memory safe API around DMA transfers.

The DMA peripheral is used to perform memory transfers in parallel to the work of the processor (the execution of the main program). A DMA transfer is more or less equivalent to spawning a thread (see `thread::spawn`) to do a `memcpy`. We'll use the fork-join model to illustrate the requirements of a memory safe API.

Consider the following DMA primitives:

```rust
/// A singleton that represents a single DMA channel (channel 1 in this case)
///
/// This singleton has exclusive access to the registers of the DMA channel 1
pub struct Dma1Channel1 {
    // ..
}

impl Dma1Channel1 {
    /// Data will be written to this `address`
    ///
    /// `inc` indicates whether the address will be incremented after every
byte transfer
    ///
    /// NOTE this performs a volatile write
    pub fn set_destination_address(&mut self, address: usize, inc: bool) {
        // ..
    }

    /// Data will be read from this `address`
    ///
    /// `inc` indicates whether the address will be incremented after every
byte transfer
    ///
    /// NOTE this performs a volatile write
    pub fn set_source_address(&mut self, address: usize, inc: bool) {
        // ..
    }

    /// Number of bytes to transfer
    ///
    /// NOTE this performs a volatile write
    pub fn set_transfer_length(&mut self, len: usize) {
        // ..
    }

    /// Starts the DMA transfer
    ///
    /// NOTE this performs a volatile write
    pub fn start(&mut self) {
        // ..
    }

    /// Stops the DMA transfer
    ///
    /// NOTE this performs a volatile write
    pub fn stop(&mut self) {
        // ..
    }

    /// Returns `true` if there's a transfer in progress
    ///
    /// NOTE this performs a volatile read
    pub fn in_progress() -> bool {
        // ..
    }
}
```

Assume that the `Dma1Channel1` is statically configured to work with serial port (AKA UART or USART) #1, `Serial1`, in one-shot mode (i.e. not circular mode). `Serial1` provides the following *blocking* API:

```
/// A singleton that represents serial port #1
pub struct Serial1 {
    // ..
}

impl Serial1 {
    /// Reads out a single byte
    ///
    /// NOTE: blocks if no byte is available to be read
    pub fn read(&mut self) -> Result<u8, Error> {
        // ..
    }

    /// Sends out a single byte
    ///
    /// NOTE: blocks if the output FIFO buffer is full
    pub fn write(&mut self, byte: u8) -> Result<(), Error> {
        // ..
    }
}
```

Let's say we want to extend `Serial1` API to (a) asynchronously send out a buffer and (b) asynchronously fill a buffer.

We'll start with a memory unsafe API and we'll iterate on it until it's completely memory safe. On each step we'll show you how the API can be broken to make you aware of the issues that need to be addressed when dealing with asynchronous memory operations.

A first stab

For starters, let's try to use the `Write::write_all` API as a reference. To keep things simple let's ignore all error handling.

```rust
/// A singleton that represents serial port #1
pub struct Serial1 {
    // NOTE: we extend this struct by adding the DMA channel singleton
    dma: Dma1Channel1,
    // ..
}

impl Serial1 {
    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn write_all<'a>(mut self, buffer: &'a [u8]) -> Transfer<&'a [u8]> {
        self.dma.set_destination_address(USART1_TX, false);
        self.dma.set_source_address(buffer.as_ptr() as usize, true);
        self.dma.set_transfer_length(buffer.len());

        self.dma.start();

        Transfer { buffer }
    }
}

/// A DMA transfer
pub struct Transfer<B> {
    buffer: B,
}

impl<B> Transfer<B> {
    /// Returns `true` if the DMA transfer has finished
    pub fn is_done(&self) -> bool {
        !Dma1Channel1::in_progress()
    }

    /// Blocks until the transfer is done and returns the buffer
    pub fn wait(self) -> B {
        // Busy wait until the transfer is done
        while !self.is_done() {}

        self.buffer
    }
}
```

NOTE: `Transfer` could expose a futures or generator based API instead of the API shown above. That's an API design question that has little bearing on the memory safety of the overall API so we won't delve into it in this text.

We can also implement an asynchronous version of `Read::read_exact`.

```
impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn read_exact<'a>(&mut self, buffer: &'a mut [u8]) -> Transfer<&'a
mut [u8]> {
        self.dma.set_source_address(USART1_RX, false);
        self.dma
            .set_destination_address(buffer.as_mut_ptr() as usize, true);
        self.dma.set_transfer_length(buffer.len());

        self.dma.start();

        Transfer { buffer }
    }
}
```

Here's how to use the `write_all` API:

```
fn write(serial: Serial1) {
    // fire and forget
    serial.write_all(b"Hello, world!\n");

    // do other stuff
}
```

And here's an example of using the `read_exact` API:

```
fn read(mut serial: Serial1) {
    let mut buf = [0; 16];
    let t = serial.read_exact(&mut buf);

    // do other stuff

    t.wait();

    match buf.split(|b| *b == b'\n').next() {
        Some(b"some-command") => { /* do something */ }
        _ => { /* do something else */ }
    }
}
```

`mem::forget`

`mem::forget` is a safe API. If our API is truly safe then we should be able to use both together without running into undefined behavior. However, that's not the case; consider the following example:

```rust
fn unsound(mut serial: Serial1) {
    start(&mut serial);
    bar();
}

#[inline(never)]
fn start(serial: &mut Serial1) {
    let mut buf = [0; 16];

    // start a DMA transfer and forget the returned `Transfer` value
    mem::forget(serial.read_exact(&mut buf));
}

#[inline(never)]
fn bar() {
    // stack variables
    let mut x = 0;
    let mut y = 0;

    // use `x` and `y`
}
```

Here we start a DMA transfer, in `start`, to fill an array allocated on the stack and then `mem::forget` the returned `Transfer` value. Then we proceed to return from `start` and execute the function `bar`.

This series of operations results in undefined behavior. The DMA transfer writes to stack memory but that memory is released when `start` returns and then reused by `bar` to allocate variables like `x` and `y`. At runtime this could result in variables `x` and `y` changing their value at random times. The DMA transfer could also overwrite the state (e.g. link register) pushed onto the stack by the prologue of function `bar`.

Note that if we had not use `mem::forget`, but `mem::drop`, it would have been possible to make `Transfer`'s destructor stop the DMA transfer and then the program would have been safe. But one can *not* rely on destructors running to enforce memory safety because `mem::forget` and memory leaks (see RC cycles) are safe in Rust.

We can fix this particular problem by changing the lifetime of the buffer from `'a` to `'static` in both APIs.

```
impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn read_exact(&mut self, buffer: &'static mut [u8]) ->
Transfer<&'static mut [u8]> {
        // .. same as before ..
    }

    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn write_all(mut self, buffer: &'static [u8]) -> Transfer<&'static
[u8]> {
        // .. same as before ..
    }
}
```

If we try to replicate the previous problem we note that `mem::forget` no longer causes problems.

```
#[allow(dead_code)]
fn sound(mut serial: Serial1, buf: &'static mut [u8; 16]) {
    // NOTE `buf` is moved into `foo`
    foo(&mut serial, buf);
    bar();
}

#[inline(never)]
fn foo(serial: &mut Serial1, buf: &'static mut [u8]) {
    // start a DMA transfer and forget the returned `Transfer` value
    mem::forget(serial.read_exact(buf));
}

#[inline(never)]
fn bar() {
    // stack variables
    let mut x = 0;
    let mut y = 0;

    // use `x` and `y`
}
```

As before, the DMA transfer continues after `mem::forget`-ing the `Transfer` value. This time that's not an issue because `buf` is statically allocated (e.g. `static mut` variable) and not on the stack.

Overlapping use

Our API doesn't prevent the user from using the `Serial` interface while the DMA transfer is in progress. This could lead the transfer to fail or data to be lost.

There are several ways to prevent overlapping use. One way is to have `Transfer` take ownership of `Serial1` and return it back when `wait` is called.

```rust
/// A DMA transfer
pub struct Transfer<B> {
    buffer: B,
    // NOTE: added
    serial: Serial1,
}

impl<B> Transfer<B> {
    /// Blocks until the transfer is done and returns the buffer
    // NOTE: the return value has changed
    pub fn wait(self) -> (B, Serial1) {
        // Busy wait until the transfer is done
        while !self.is_done() {}

        (self.buffer, self.serial)
    }

    // ..
}

impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    // NOTE we now take `self` by value
    pub fn read_exact(mut self, buffer: &'static mut [u8]) ->
Transfer<&'static mut [u8]> {
        // .. same as before ..

        Transfer {
            buffer,
            // NOTE: added
            serial: self,
        }
    }

    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    // NOTE we now take `self` by value
    pub fn write_all(mut self, buffer: &'static [u8]) -> Transfer<&'static
[u8]> {
        // .. same as before ..

        Transfer {
            buffer,
            // NOTE: added
            serial: self,
        }
    }
}
```

The move semantics statically prevent access to `Serial1` while the transfer is in progress.

```
fn read(serial: Serial1, buf: &'static mut [u8; 16]) {
    let t = serial.read_exact(buf);

    // let byte = serial.read(); //~ ERROR: `serial` has been moved

    // .. do stuff ..

    let (serial, buf) = t.wait();

    // .. do more stuff ..
}
```

There are other ways to prevent overlapping use. For example, a (`Cell`) flag that indicates whether a DMA transfer is in progress could be added to `Serial1` . When the flag is set `read` , `write` , `read_exact` and `write_all` would all return an error (e.g. `Error::InUse`) at runtime. The flag would be set when `write_all` / `read_exact` is used and cleared in `Transfer.wait` .

Compiler (mis)optimizations

The compiler is free to re-order and merge non-volatile memory operations to better optimize a program. With our current API, this freedom can lead to undefined behavior. Consider the following example:

```
fn reorder(serial: Serial1, buf: &'static mut [u8]) {
    // zero the buffer (for no particular reason)
    buf.iter_mut().for_each(|byte| *byte = 0);

    let t = serial.read_exact(buf);

    // ... do other stuff ..

    let (buf, serial) = t.wait();

    buf.reverse();

    // .. do stuff with `buf` ..
}
```

Here the compiler is free to move `buf.reverse()` before `t.wait()` , which would result in a data race: both the processor and the DMA would end up modifying `buf` at the same time. Similarly the compiler can move the zeroing operation to after `read_exact` , which would also result in a data race.

To prevent these problematic reorderings we can use a `compiler_fence`

```
impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn read_exact(mut self, buffer: &'static mut [u8]) ->
Transfer<&'static mut [u8]> {
        self.dma.set_source_address(USART1_RX, false);
        self.dma
            .set_destination_address(buffer.as_mut_ptr() as usize, true);
        self.dma.set_transfer_length(buffer.len());

        // NOTE: added
        atomic::compiler_fence(Ordering::Release);

        // NOTE: this is a volatile *write*
        self.dma.start();

        Transfer {
            buffer,
            serial: self,
        }
    }

    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn write_all(mut self, buffer: &'static [u8]) -> Transfer<&'static
[u8]> {
        self.dma.set_destination_address(USART1_TX, false);
        self.dma.set_source_address(buffer.as_ptr() as usize, true);
        self.dma.set_transfer_length(buffer.len());

        // NOTE: added
        atomic::compiler_fence(Ordering::Release);

        // NOTE: this is a volatile *write*
        self.dma.start();

        Transfer {
            buffer,
            serial: self,
        }
    }
}

impl<B> Transfer<B> {
    /// Blocks until the transfer is done and returns the buffer
    pub fn wait(self) -> (B, Serial1) {
        // NOTE: this is a volatile *read*
        while !self.is_done() {}

        // NOTE: added
        atomic::compiler_fence(Ordering::Acquire);

        (self.buffer, self.serial)
    }
    // ..
}
```

We use `Ordering::Release` in `read_exact` and `write_all` to prevent all preceding memory operations from being moved *after* `self.dma.start()`, which performs a volatile write.

Likewise, we use `Ordering::Acquire` in `Transfer.wait` to prevent all subsequent memory operations from being moved *before* `self.is_done()`, which performs a volatile read.

To better visualize the effect of the fences here's a slightly tweaked version of the example from the previous section. We have added the fences and their orderings in the comments.

```
fn reorder(serial: Serial1, buf: &'static mut [u8], x: &mut u32) {
    // zero the buffer (for no particular reason)
    buf.iter_mut().for_each(|byte| *byte = 0);

    *x += 1;

    let t = serial.read_exact(buf); // compiler_fence(Ordering::Release) ▲

    // NOTE: the processor can't access `buf` between the fences
    // ... do other stuff ..
    *x += 2;

    let (buf, serial) = t.wait(); // compiler_fence(Ordering::Acquire) ▼

    *x += 3;

    buf.reverse();

    // .. do stuff with `buf` ..
}
```

The zeroing operation can *not* be moved *after* `read_exact` due to the `Release` fence. Similarly, the `reverse` operation can *not* be moved *before* `wait` due to the `Acquire` fence. The memory operations *between* both fences *can* be freely reordered across the fences but none of those operations involves `buf` so such reorderings do *not* result in undefined behavior.

Note that `compiler_fence` is a bit stronger than what's required. For example, the fences will prevent the operations on `x` from being merged even though we know that `buf` doesn't overlap with `x` (due to Rust aliasing rules). However, there exist no intrinsic that's more fine grained than `compiler_fence`.

Don't we need a memory barrier?

That depends on the target architecture. In the case of Cortex M0 to M4F cores, AN321 says:

> 3.2 Typical usages
>
> (..)
>
> The use of DMB is rarely needed in Cortex-M processors because they do not reorder memory transactions. However, it is needed if the software is to be reused on other ARM processors, especially multi-master systems. For example:
>
> - DMA controller configuration. A barrier is required between a CPU memory access and a DMA operation.
>
> (..)
>
> 4.18 Multi-master systems
>
> (..)
>
> Omitting the DMB or DSB instruction in the examples in Figure 41 on page 47 and Figure 42 would not cause any error because the Cortex-M processors:
>
> - do not re-order memory transfers
> - do not permit two write transfers to be overlapped.

Where Figure 41 shows a DMB (memory barrier) instruction being used before starting a DMA transaction.

In the case of Cortex-M7 cores you'll need memory barriers (DMB/DSB) if you are using the data cache (DCache), unless you manually invalidate the buffer used by the DMA. Even with the data cache disabled, memory barriers might still be required to avoid reordering in the store buffer.

If your target is a multi-core system then it's very likely that you'll need memory barriers.

If you do need the memory barrier then you need to use `atomic::fence` instead of `compiler_fence`. That should generate a DMB instruction on Cortex-M devices.

Generic buffer

Our API is more restrictive that it needs to be. For example, the following program won't be accepted even though it's valid.

```
fn reuse(serial: Serial1, msg: &'static mut [u8]) {
    // send a message
    let t1 = serial.write_all(msg);

    // ..

    let (msg, serial) = t1.wait(); // `msg` is now `&'static [u8]`

    msg.reverse();

    // now send it in reverse
    let t2 = serial.write_all(msg);

    // ..

    let (buf, serial) = t2.wait();

    // ..
}
```

To accept such program we can make the buffer argument generic.

```rust
// as-slice = "0.1.0"
use as_slice::{AsMutSlice, AsSlice};

impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn read_exact<B>(mut self, mut buffer: B) -> Transfer<B>
    where
        B: AsMutSlice<Element = u8>,
    {
        // NOTE: added
        let slice = buffer.as_mut_slice();
        let (ptr, len) = (slice.as_mut_ptr(), slice.len());

        self.dma.set_source_address(USART1_RX, false);

        // NOTE: tweaked
        self.dma.set_destination_address(ptr as usize, true);
        self.dma.set_transfer_length(len);

        atomic::compiler_fence(Ordering::Release);
        self.dma.start();

        Transfer {
            buffer,
            serial: self,
        }
    }

    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    fn write_all<B>(mut self, buffer: B) -> Transfer<B>
    where
        B: AsSlice<Element = u8>,
    {
        // NOTE: added
        let slice = buffer.as_slice();
        let (ptr, len) = (slice.as_ptr(), slice.len());

        self.dma.set_destination_address(USART1_TX, false);

        // NOTE: tweaked
        self.dma.set_source_address(ptr as usize, true);
        self.dma.set_transfer_length(len);

        atomic::compiler_fence(Ordering::Release);
        self.dma.start();

        Transfer {
            buffer,
            serial: self,
        }
    }
}
```

NOTE: `AsRef<[u8]>` (`AsMut<[u8]>`) could have been used instead of `AsSlice<Element = u8>` (`AsMutSlice<Element = u8`).

Now the `reuse` program will be accepted.

Immovable buffers

With this modification the API will also accept arrays by value (e.g. `[u8; 16]`). However, using arrays can result in pointer invalidation. Consider the following program.

```
fn invalidate(serial: Serial1) {
    let t = start(serial);

    bar();

    let (buf, serial) = t.wait();
}

#[inline(never)]
fn start(serial: Serial1) -> Transfer<[u8; 16]> {
    // array allocated in this frame
    let buffer = [0; 16];

    serial.read_exact(buffer)
}

#[inline(never)]
fn bar() {
    // stack variables
    let mut x = 0;
    let mut y = 0;

    // use `x` and `y`
}
```

The `read_exact` operation will use the address of the `buffer` local to the `start` function. That local `buffer` will be freed when `start` returns and the pointer used in `read_exact` will become invalidated. You'll end up with a situation similar to the `unsound` example.

To avoid this problem we require that the buffer used with our API retains its memory location even when it's moved. The `Pin` newtype provides such guarantee. We can update our API to required that all buffers are "pinned" first.

> **NOTE:** To compile all the programs below this point you'll need Rust `>=1.33.0`. As of time of writing (2019-01-04) that means using the nightly channel.

```
/// A DMA transfer
pub struct Transfer<B> {
    // NOTE: changed
    buffer: Pin<B>,
    serial: Serial1,
}

impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn read_exact<B>(mut self, mut buffer: Pin<B>) -> Transfer<B>
    where
        // NOTE: bounds changed
        B: DerefMut,
        B::Target: AsMutSlice<Element = u8> + Unpin,
    {
        // .. same as before ..
    }

    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn write_all<B>(mut self, buffer: Pin<B>) -> Transfer<B>
    where
        // NOTE: bounds changed
        B: Deref,
        B::Target: AsSlice<Element = u8>,
    {
        // .. same as before ..
    }
}
```

> **NOTE:** We could have used the `StableDeref` trait instead of the `Pin` newtype but opted for `Pin` since it's provided in the standard library.

With this new API we can use `&'static mut` references, `Box`-ed slices, `Rc`-ed slices, etc.

```rust
fn static_mut(serial: Serial1, buf: &'static mut [u8]) {
    let buf = Pin::new(buf);

    let t = serial.read_exact(buf);

    // ..

    let (buf, serial) = t.wait();

    // ..
}

fn boxed(serial: Serial1, buf: Box<[u8]>) {
    let buf = Pin::new(buf);

    let t = serial.read_exact(buf);

    // ..

    let (buf, serial) = t.wait();

    // ..
}
```

`'static` bound

Does pinning let us safely use stack allocated arrays? The answer is *no*. Consider the following example.

```
fn unsound(serial: Serial1) {
    start(serial);

    bar();
}

// pin-utils = "0.1.0-alpha.4"
use pin_utils::pin_mut;

#[inline(never)]
fn start(serial: Serial1) {
    let buffer = [0; 16];

    // pin the `buffer` to this stack frame
    // `buffer` now has type `Pin<&mut [u8; 16]>`
    pin_mut!(buffer);

    mem::forget(serial.read_exact(buffer));
}

#[inline(never)]
fn bar() {
    // stack variables
    let mut x = 0;
    let mut y = 0;

    // use `x` and `y`
}
```

As seen many times before, the above program runs into undefined behavior due to stack frame corruption.

The API is unsound for buffers of type `Pin<&'a mut [u8]>` where `'a` is *not* `'static`. To prevent the problem we have to add a `'static` bound in some places.

```rust
impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn read_exact<B>(mut self, mut buffer: Pin<B>) -> Transfer<B>
    where
        // NOTE: added 'static bound
        B: DerefMut + 'static,
        B::Target: AsMutSlice<Element = u8> + Unpin,
    {
        // .. same as before ..
    }

    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn write_all<B>(mut self, buffer: Pin<B>) -> Transfer<B>
    where
        // NOTE: added 'static bound
        B: Deref + 'static,
        B::Target: AsSlice<Element = u8>,
    {
        // .. same as before ..
    }
}
```

Now the problematic program will be rejected.

Destructors

Now that the API accepts `Box` -es and other types that have destructors we need to decide what to do when `Transfer` is early-dropped.

Normally, `Transfer` values are consumed using the `wait` method but it's also possible to, implicitly or explicitly, `drop` the value before the transfer is over. For example, dropping a `Transfer<Box<[u8]>>` value will cause the buffer to be deallocated. This can result in undefined behavior if the transfer is still in progress as the DMA would end up writing to deallocated memory.

In such scenario one option is to make `Transfer.drop` stop the DMA transfer. The other option is to make `Transfer.drop` wait for the transfer to finish. We'll pick the former option as it's cheaper.

```
/// A DMA transfer
pub struct Transfer<B> {
    // NOTE: always `Some` variant
    inner: Option<Inner<B>>,
}

// NOTE: previously named `Transfer<B>`
struct Inner<B> {
    buffer: Pin<B>,
    serial: Serial1,
}

impl<B> Transfer<B> {
    /// Blocks until the transfer is done and returns the buffer
    pub fn wait(mut self) -> (Pin<B>, Serial1) {
        while !self.is_done() {}

        atomic::compiler_fence(Ordering::Acquire);

        let inner = self
            .inner
            .take()
            .unwrap_or_else(|| unsafe { hint::unreachable_unchecked() });
        (inner.buffer, inner.serial)
    }
}

impl<B> Drop for Transfer<B> {
    fn drop(&mut self) {
        if let Some(inner) = self.inner.as_mut() {
            // NOTE: this is a volatile write
            inner.serial.dma.stop();

            // we need a read here to make the Acquire fence effective
            // we do *not* need this if `dma.stop` does a RMW operation
            unsafe {
                ptr::read_volatile(&0);
            }

            // we need a fence here for the same reason we need one in
`Transfer.wait`
            atomic::compiler_fence(Ordering::Acquire);
        }
    }
}

impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn read_exact<B>(mut self, mut buffer: Pin<B>) -> Transfer<B>
    where
        B: DerefMut + 'static,
        B::Target: AsMutSlice<Element = u8> + Unpin,
    {
        // .. same as before ..
        Transfer {
            inner: Some(Inner {
                buffer,
                serial: self,
            }),
        }
    }

    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn write_all<B>(mut self, buffer: Pin<B>) -> Transfer<B>
    where
        B: Deref + 'static,
        B::Target: AsSlice<Element = u8>,
    {
        // .. same as before ..

        Transfer {
            inner: Some(Inner {
                buffer,
                serial: self,
            }),
        }
    }
}
```

Now the DMA transfer will be stopped before the buffer is deallocated.

```
fn reuse(serial: Serial1) {
    let buf = Pin::new(Box::new([0; 16]));

    let t = serial.read_exact(buf); // compiler_fence(Ordering::Release) ▲

    // ..

    // this stops the DMA transfer and frees memory
    mem::drop(t); // compiler_fence(Ordering::Acquire) ▼

    // this likely reuses the previous memory allocation
    let mut buf = Box::new([0; 16]);

    // .. do stuff with `buf` ..
}
```

Summary

To sum it up, we need to consider all the following points to achieve memory safe DMA transfers:

- Use immovable buffers plus indirection: `Pin<B>`. Alternatively, you can use the `StableDeref` trait.
- The ownership of the buffer must be passed to the DMA : `B: 'static`.
- Do *not* rely on destructors running for memory safety. Consider what happens if `mem::forget` is used with your API.
- *Do* add a custom destructor that stops the DMA transfer, or waits for it to finish. Consider what happens if `mem::drop` is used with your API.

This text leaves out up several details required to build a production grade DMA abstraction, like configuring the DMA channels (e.g. streams, circular vs one-shot mode, etc.), alignment of buffers, error handling, how to make the abstraction device-agnostic, etc. All those aspects are left as an exercise for the reader / community (`:P`).

A note on compiler support

This book makes use of a built-in *compiler* target, the `thumbv7m-none-eabi`, for which the Rust team distributes a `rust-std` component, which is a pre-compiled collection of crates like `core` and `std`.

If you want to attempt replicating the contents of this book for a different target architecture, you need to take into account the different levels of support that Rust provides for (compilation) targets.

LLVM support

As of Rust 1.28, the official Rust compiler, `rustc`, uses LLVM for (machine) code generation. The minimal level of support Rust provides for an architecture is having its LLVM backend enabled in `rustc`. You can see all the architectures that `rustc` supports, through LLVM, by running the following command:

```
$ # you need to have `cargo-binutils` installed to run this command
$ cargo objdump -- -version
LLVM (http://llvm.org/):
  LLVM version 7.0.0svn
  Optimized build.
  Default target: x86_64-unknown-linux-gnu
  Host CPU: skylake

  Registered Targets:
    aarch64    - AArch64 (little endian)
    aarch64_be - AArch64 (big endian)
    arm        - ARM
    arm64      - ARM64 (little endian)
    armeb      - ARM (big endian)
    hexagon    - Hexagon
    mips       - Mips
    mips64     - Mips64 [experimental]
    mips64el   - Mips64el [experimental]
    mipsel     - Mipsel
    msp430     - MSP430 [experimental]
    nvptx      - NVIDIA PTX 32-bit
    nvptx64    - NVIDIA PTX 64-bit
    ppc32      - PowerPC 32
    ppc64      - PowerPC 64
    ppc64le    - PowerPC 64 LE
    sparc      - Sparc
    sparcel    - Sparc LE
    sparcv9    - Sparc V9
    systemz    - SystemZ
    thumb      - Thumb
    thumbeb    - Thumb (big endian)
    wasm32     - WebAssembly 32-bit
    wasm64     - WebAssembly 64-bit
    x86        - 32-bit X86: Pentium-Pro and above
    x86-64     - 64-bit X86: EM64T and AMD64
```

If LLVM supports the architecture you are interested in, but `rustc` is built with the backend disabled (which is the case of AVR as of Rust 1.28), then you will need to modify the Rust source enabling it. The first two commits of PR rust-lang/rust#52787 give you an idea of the required changes.

On the other hand, if LLVM doesn't support the architecture, but a fork of LLVM does, you will have to replace the original version of LLVM with the fork before building `rustc`. The Rust build system allows this and in principle it should just require changing the `llvm` submodule to point to the fork.

If your target architecture is only supported by some vendor provided GCC, you have the option of using `mrustc`, an unofficial Rust compiler, to translate your Rust program into C code and then compile that using GCC.

Built-in target

A compilation target is more than just its architecture. Each target has a specification associated to it that describes, among other things, its architecture, its operating system and the default linker.

The Rust compiler knows about several targets. These are *built into* the compiler and can be listed by running the following command:

```
$ rustc --print target-list | column
aarch64-fuchsia                   mipsisa32r6el-unknown-linux-gnu
aarch64-linux-android             mipsisa64r6-unknown-linux-gnuabi64
aarch64-pc-windows-msvc           mipsisa64r6el-unknown-linux-gnuabi64
aarch64-unknown-cloudabi          msp430-none-elf
aarch64-unknown-freebsd           nvptx64-nvidia-cuda
aarch64-unknown-hermit            powerpc-unknown-linux-gnu
aarch64-unknown-linux-gnu         powerpc-unknown-linux-gnuspe
aarch64-unknown-linux-musl        powerpc-unknown-linux-musl
aarch64-unknown-netbsd            powerpc-unknown-netbsd
aarch64-unknown-none              powerpc-wrs-vxworks
aarch64-unknown-none-softfloat    powerpc-wrs-vxworks-spe
aarch64-unknown-openbsd           powerpc64-unknown-freebsd
aarch64-unknown-redox             powerpc64-unknown-linux-gnu
aarch64-uwp-windows-msvc          powerpc64-unknown-linux-musl
aarch64-wrs-vxworks               powerpc64-wrs-vxworks
arm-linux-androideabi             powerpc64le-unknown-linux-gnu
arm-unknown-linux-gnueabi         powerpc64le-unknown-linux-musl
arm-unknown-linux-gnueabihf       riscv32i-unknown-none-elf
arm-unknown-linux-musleabi        riscv32imac-unknown-none-elf
arm-unknown-linux-musleabihf      riscv32imc-unknown-none-elf
armebv7r-none-eabi                riscv64gc-unknown-linux-gnu
armebv7r-none-eabihf              riscv64gc-unknown-none-elf
armv4t-unknown-linux-gnueabi      riscv64imac-unknown-none-elf
armv5te-unknown-linux-gnueabi     s390x-unknown-linux-gnu
armv5te-unknown-linux-musleabi    sparc-unknown-linux-gnu
armv6-unknown-freebsd             sparc64-unknown-linux-gnu
armv6-unknown-netbsd-eabihf       sparc64-unknown-netbsd
armv7-linux-androideabi           sparc64-unknown-openbsd
armv7-unknown-cloudabi-eabihf     sparcv9-sun-solaris
armv7-unknown-freebsd             thumbv6m-none-eabi
armv7-unknown-linux-gnueabi       thumbv7a-pc-windows-msvc
armv7-unknown-linux-gnueabihf     thumbv7em-none-eabi
armv7-unknown-linux-musleabi      thumbv7em-none-eabihf
armv7-unknown-linux-musleabihf    thumbv7m-none-eabi
armv7-unknown-netbsd-eabihf       thumbv7neon-linux-androideabi
armv7-wrs-vxworks-eabihf          thumbv7neon-unknown-linux-gnueabihf
armv7a-none-eabi                  thumbv7neon-unknown-linux-musleabihf
armv7a-none-eabihf                thumbv8m.base-none-eabi
armv7r-none-eabi                  thumbv8m.main-none-eabi
armv7r-none-eabihf                thumbv8m.main-none-eabihf
asmjs-unknown-emscripten          wasm32-unknown-emscripten
hexagon-unknown-linux-musl        wasm32-unknown-unknown
i586-pc-windows-msvc              wasm32-wasi
i586-unknown-linux-gnu            x86_64-apple-darwin
i586-unknown-linux-musl           x86_64-fortanix-unknown-sgx
i686-apple-darwin                 x86_64-fuchsia
i686-linux-android                x86_64-linux-android
i686-pc-windows-gnu               x86_64-linux-kernel
i686-pc-windows-msvc              x86_64-pc-solaris
i686-unknown-cloudabi             x86_64-pc-windows-gnu
i686-unknown-freebsd              x86_64-pc-windows-msvc
i686-unknown-haiku                x86_64-rumprun-netbsd
i686-unknown-linux-gnu            x86_64-sun-solaris
i686-unknown-linux-musl           x86_64-unknown-cloudabi
i686-unknown-netbsd               x86_64-unknown-dragonfly
```

```
i686-unknown-openbsd              x86_64-unknown-freebsd
i686-unknown-uefi                 x86_64-unknown-haiku
i686-uwp-windows-gnu              x86_64-unknown-hermit
i686-uwp-windows-msvc             x86_64-unknown-hermit-kernel
i686-wrs-vxworks                  x86_64-unknown-illumos
mips-unknown-linux-gnu            x86_64-unknown-l4re-uclibc
mips-unknown-linux-musl           x86_64-unknown-linux-gnu
mips-unknown-linux-uclibc         x86_64-unknown-linux-gnux32
mips64-unknown-linux-gnuabi64     x86_64-unknown-linux-musl
mips64-unknown-linux-muslabi64    x86_64-unknown-netbsd
mips64el-unknown-linux-gnuabi64   x86_64-unknown-openbsd
mips64el-unknown-linux-muslabi64  x86_64-unknown-redox
mipsel-unknown-linux-gnu          x86_64-unknown-uefi
mipsel-unknown-linux-musl         x86_64-uwp-windows-gnu
mipsel-unknown-linux-uclibc       x86_64-uwp-windows-msvc
mipsisa32r6-unknown-linux-gnu     x86_64-wrs-vxworks
```

You can print the specification of one of these targets using the following command:

```
$ rustc +nightly -Z unstable-options --print target-spec-json --target
thumbv7m-none-eabi
{
  "abi-blacklist": [
    "stdcall",
    "fastcall",
    "vectorcall",
    "thiscall",
    "win64",
    "sysv64"
  ],
  "arch": "arm",
  "data-layout": "e-m:e-p:32:32-i64:64-v128:64:128-a:0:32-n32-S64",
  "emit-debug-gdb-scripts": false,
  "env": "",
  "executables": true,
  "is-builtin": true,
  "linker": "arm-none-eabi-gcc",
  "linker-flavor": "gcc",
  "llvm-target": "thumbv7m-none-eabi",
  "max-atomic-width": 32,
  "os": "none",
  "panic-strategy": "abort",
  "relocation-model": "static",
  "target-c-int-width": "32",
  "target-endian": "little",
  "target-pointer-width": "32",
  "vendor": ""
}
```

If none of these built-in targets seems appropriate for your target system, you'll have to create a custom target by writing your own target specification file in JSON format which is described in the next section.

`rust-std` component

For some of the built-in target the Rust team distributes `rust-std` components via `rustup`. This component is a collection of pre-compiled crates like `core` and `std`, and it's required for cross compilation.

You can find the list of targets that have a `rust-std` component available via `rustup` by running the following command:

```
$ rustup target list | column
aarch64-apple-ios                       mipsel-unknown-linux-musl
aarch64-fuchsia                         nvptx64-nvidia-cuda
aarch64-linux-android                   powerpc-unknown-linux-gnu
aarch64-pc-windows-msvc                 powerpc64-unknown-linux-gnu
aarch64-unknown-linux-gnu               powerpc64le-unknown-linux-gnu
aarch64-unknown-linux-musl              riscv32i-unknown-none-elf
aarch64-unknown-none                    riscv32imac-unknown-none-elf
aarch64-unknown-none-softfloat          riscv32imc-unknown-none-elf
arm-linux-androideabi                   riscv64gc-unknown-linux-gnu
arm-unknown-linux-gnueabi               riscv64gc-unknown-none-elf
arm-unknown-linux-gnueabihf             riscv64imac-unknown-none-elf
arm-unknown-linux-musleabi              s390x-unknown-linux-gnu
arm-unknown-linux-musleabihf            sparc64-unknown-linux-gnu
armebv7r-none-eabi                      sparcv9-sun-solaris
armebv7r-none-eabihf                    thumbv6m-none-eabi
armv5te-unknown-linux-gnueabi           thumbv7em-none-eabi
armv5te-unknown-linux-musleabi          thumbv7em-none-eabihf
armv7-linux-androideabi                 thumbv7m-none-eabi
armv7-unknown-linux-gnueabi             thumbv7neon-linux-androideabi
armv7-unknown-linux-gnueabihf           thumbv7neon-unknown-linux-gnueabihf
armv7-unknown-linux-musleabi            thumbv8m.base-none-eabi
armv7-unknown-linux-musleabihf          thumbv8m.main-none-eabi
armv7a-none-eabi                        thumbv8m.main-none-eabihf
armv7r-none-eabi                        wasm32-unknown-emscripten
armv7r-none-eabihf                      wasm32-unknown-unknown
asmjs-unknown-emscripten                wasm32-wasi
i586-pc-windows-msvc                    x86_64-apple-darwin
i586-unknown-linux-gnu                  x86_64-apple-ios
i586-unknown-linux-musl                 x86_64-fortanix-unknown-sgx
i686-linux-android                      x86_64-fuchsia
i686-pc-windows-gnu                     x86_64-linux-android
i686-pc-windows-msvc                    x86_64-pc-windows-gnu
i686-unknown-freebsd                    x86_64-pc-windows-msvc
i686-unknown-linux-gnu                  x86_64-rumprun-netbsd
i686-unknown-linux-musl                 x86_64-sun-solaris
mips-unknown-linux-gnu                  x86_64-unknown-cloudabi
mips-unknown-linux-musl                 x86_64-unknown-freebsd
mips64-unknown-linux-gnuabi64           x86_64-unknown-linux-gnu (default)
mips64-unknown-linux-muslabi64          x86_64-unknown-linux-gnux32
mips64el-unknown-linux-gnuabi64         x86_64-unknown-linux-musl
mips64el-unknown-linux-muslabi64        x86_64-unknown-netbsd
mipsel-unknown-linux-gnu                x86_64-unknown-redox
```

If there's no `rust-std` component for your target, or you are using a custom target, then you'll have to use a nightly toolchain to build the standard library. See the next page about building for custom targets.

Creating a custom target

If a custom target triple is not available for your platform, you must create a custom target file that describes your target to rustc.

Keep in mind that it is required to use a nightly compiler to build the core library, which must be done for a target unknown to rustc.

Deciding on a target triple

Many targets already have a known triple used to describe them, typically in the form ARCH-VENDOR-SYS-ABI. You should aim to use the same triple that LLVM uses; however, it may differ if you need to specify additional information to Rust that LLVM does not know about. Although the triple is technically only for human use, it's important for it to be unique and descriptive especially if the target will be upstreamed in the future.

The ARCH part is typically just the architecture name, except in the case of 32-bit ARM. For example, you would probably use x86_64 for those processors, but specify the exact ARM architecture version. Typical values might be `armv7` , `armv5te` , or `thumbv7neon` . Take a look at the names of the built-in targets for inspiration.

The VENDOR part is optional and describes the manufacturer. Omitting this field is the same as using `unknown` .

The SYS part describes the OS that is used. Typical values include `win32` , `linux` , and `darwin` for desktop platforms. `none` is used for bare-metal usage.

The ABI part describes how the process starts up. `eabi` is used for bare metal, while `gnu` is used for glibc, `musl` for musl, etc.

Now that you have a target triple, create a file with the name of the triple and a `.json` extension. For example, a file describing `armv7a-none-eabi` would have the filename `armv7a-none-eabi.json` .

Fill the target file

The target file must be valid JSON. There are two places where its contents are described: `Target` , where every field is mandatory, and `TargetOptions` , where every field is optional. **All underscores are replaced with hyphens**.

The recommended way is to base your target file on the specification of a built-in target that's similar to your target system, then tweak it to match the properties of your target system. To do so, use the command `rustc +nightly -Z unstable-options --print target-spec-json --target $SOME_SIMILAR_TARGET`, using a target that's already built into the compiler.

You can pretty much copy that output into your file. Start with a few modifications:

- Remove `"is-builtin": true`
- Fill `llvm-target` with the triple that LLVM expects
- Decide on a panicking strategy. A bare metal implementation will likely use `"panic-strategy": "abort"`. If you decide not to `abort` on panicking, unless you tell Cargo to per-project, you must define an eh_personality function.
- Configure atomics. Pick the first option that describes your target:
  - I have a single-core processor, no threads, **no interrupts**, or any way for multiple things to be happening in parallel: if you are **sure** that is the case, such as WASM (for now), you may set `"singlethread": true`. This will configure LLVM to convert all atomic operations to use their single threaded counterparts. Incorrectly using this option may result in UB if using threads or interrupts.
  - I have native atomic operations: set `max-atomic-width` to the biggest type in bits that your target can operate on atomically. For example, many ARM cores have 32-bit atomic operations. You may set `"max-atomic-width": 32` in that case.
  - I have no native atomic operations, but I can emulate them myself: set `max-atomic-width` to the highest number of bits that you can emulate up to 128, then implement all of the atomic and sync functions expected by LLVM as `#[no_mangle] unsafe extern "C"`. These functions have been standardized by gcc, so the gcc documentation may have more notes. Missing functions will cause a linker error, while incorrectly implemented functions will possibly cause UB. For example, if you have a single-core, single-thread processor with interrupts, you can implement these functions to disable interrupts, perform the regular operation, and then re-enable them.
  - I have no native atomic operations: you'll have to do some unsafe work to manually ensure synchronization in your code. You must set `"max-atomic-width": 0`.
- Change the linker if integrating with an existing toolchain. For example, if you're using a toolchain that uses a custom build of gcc, set `"linker-flavor": "gcc"` and `linker` to the command name of your linker. If you require additional linker arguments, use `pre-link-args` and `post-link-args` as so:

```
"pre-link-args": {
    "gcc": [
        "-Wl,--as-needed",
        "-Wl,-z,noexecstack",
        "-m64"
    ]
},
"post-link-args": {
    "gcc": [
        "-Wl,--allow-multiple-definition",
        "-Wl,--start-group,-lc,-lm,-lgcc,-lstdc++,-lsupc++,--end-group"
    ]
}
```

Ensure that the linker type is the key within `link-args`.

- Configure LLVM features. Run `llc -march=ARCH -mattr=help` where ARCH is the base architecture (not including the version in the case of ARM) to list the available features and their descriptions. **If your target requires strict memory alignment access (e.g. `armv5te`), make sure that you enable `strict-align`**. To enable a feature, place a plus before it. Likewise, to disable a feature, place a minus before it. Features should be comma separated like so: `"features": "+soft-float,+neon`. Note that this may not be necessary if LLVM knows enough about your target based on the provided triple and CPU.
- Configure the CPU that LLVM uses if you know it. This will enable CPU-specific optimizations and features. At the top of the output of the command in the last step, there is a list of known CPUs. If you know that you will be targeting a specific CPU, you may set it in the `cpu` field in the JSON target file.

Use the target file

Once you have a target specification file, you may refer to it by its path or by its name (i.e. excluding `.json`) if it is in the current directory or in `$RUST_TARGET_PATH`.

Verify that it is readable by rustc:

```
❯ rustc --print cfg --target foo.json # or just foo if in the current
directory
debug_assertions
target_arch="arm"
target_endian="little"
target_env=""
target_feature="mclass"
target_feature="v7"
target_has_atomic="16"
target_has_atomic="32"
target_has_atomic="8"
target_has_atomic="cas"
target_has_atomic="ptr"
target_os="none"
target_pointer_width="32"
target_vendor=""
```

Now, you finally get to use it! Many resources have been recommending `xargo` or `cargo-xbuild`. However, its successor, cargo's `build-std` feature, has received a lot of work recently and has quickly reached feature parity with the other options. As such, this guide will only cover that option.

Start with a bare minimum `no_std` program. Now, run `cargo build -Z build-std=core --target foo.json`, again using the above rules about referencing the path. Hopefully, you should now have a binary in the target directory.

You may optionally configure cargo to always use your target. See the recommendations at the end of the page about the smallest `no_std` program. However, you'll currently have to use the flag `-Z build-std=core` as that option is unstable.

Build additional built-in crates

When using cargo's `build-std` feature, you can choose which crates to compile in. By default, when only passing `-Z build-std`, `std`, `core`, and `alloc` are compiled. However, you may want to exclude `std` when compiling for bare-metal. To do so, specify the crated you'd like after `build-std`. For example, to include `core` and `alloc`, pass `-Z build-std=core,alloc`.

Troubleshooting

language item required, but not found: `eh_personality`

Either add `"panic-strategy": "abort"` to your target file, or define an eh_personality function. Alternatively, tell Cargo to ignore it.

undefined reference to `__sync_val_compare_and_swap_#`

Rust thinks that your target has atomic instructions, but LLVM doesn't. Go back to the step about configuring atomics. You will need to reduce the number in `max-atomic-width`. See #58500 for more details.

could not find `sync` in `alloc`

Similar to the above case, Rust doesn't think that you have atomics. You must implement them yourself or tell Rust that you have atomic instructions.

multiple definition of `__(something)`

You're likely linking your Rust program with code built from another language, and the other language includes compiler built-ins that Rust also creates. To fix this, you'll need to tell your linker to allow multiple definitions. If using gcc, you may add:

```
"post-link-args": {
    "gcc": [
        "-Wl,--allow-multiple-definition"
    ]
}
```

error adding symbols: file format not recognized

Switch to cargo's `build-std` feature and update your compiler. This was a bug introduced for a few compiler builds that tried to pass in internal Rust object to an external linker.